Good morning Eclipse family. My name is Kim Moir and I’m a release engineer at Mozilla. I’m also an Eclipse release engineer alumni. Today I’m going to discuss how Mozilla scales their infrastructure to build and test software at tremendous scale. Also, I’ll touch on how we manage this from a human perspective, because that isn’t easy either. I’ll conclude with some lessons learned and how these can apply to your environment. I’ll be happy to answer any questions you may have at the end.

---

References

toolbox picture

http://www.flickr.com/photos/mtsofan/9837413583/sizes/l/
You’re probably familiar with the products we build, such as Firefox for Desktop and Android and Firefox OS. Firefox OS is a relatively new product that Mozilla started working on a few years ago. It’s an open source operating system for smartphones. When there was a new product coming on line, we knew that we would have to be able to scale our build farm to handle additional load.

So you’re probably familiar with the products, but not aware of what it takes to build and test them.

As I was preparing this talk I realized that there’s probably not a lot of people in the audience that are familiar with how we do things at Mozilla. So I’m going to start simply with three things you should know about Mozilla release engineering.

Note that we ship Firefox on four platforms and with 90+ locales on the same day as US English.
# Daily

6000 build jobs

50,000 test jobs

#1 We run a lot of builds and tests.

At Mozilla we land code, at Eclipse the term is to commit it. Each time a developer lands a change, it invokes a series of builds and associated tests on relevant platforms. Within each test job there are many actual test suites that run.

References

From Armen

6,000 build jobs/weekday * 5 * 52 = 1,560,000 (even with rounding down and excluding weekends)

50,000 test jobs/weekday * 5 * 52 = 13,000,000 (we rarely have less than 50k test jobs on a week day)

Assuming that 10 hours per push is still "the number" for every check-in then:

10 hours/push * 80,000 pushes in 2013 = 800,000 hours = 2192 days = 91
We have a commitment to developers that jobs should start within 15 minutes of being requested. We don’t have a perfect record on this, but certainly our numbers are good. We have metrics that measure this every day so we can see what platforms need additional capacity. And we adjust capacity as needed, and remove old platforms as they become less relevant in the marketplace.
Devices

- 5000+ in total
- 1800+ for builds, 3900+ for tests
- Windows, Mac, Linux and Android
- 80% of build device pool in AWS, 50% of test

#2 We have lot of hardware used on our build farm, both in our datacenters, and virtually (AWS)

This 80% number does not reflect the amount of traffic just the amount of available devices

We are able to have more build machines than tests in AWS because we need to run tests on the actual OS and hardware that users experience. There aren’t any mac or windows (desktop) images in AWS. We also run Android tests on rack mounted reference boards for some platforms, some on emulators.

References
https://secure.pub.build.mozilla.org/builddata/reports/slave_health/index.html
* https://secure.pub.build.mozilla.org/slavealloc/ui/#silos
#3 The Mozilla community spans the globe so it is only fitting that we have an international group of release engineers. Here we are at a workweek in Boston wearing our ship it shirts. I’m wearing mine today.

Why does Mozilla invest so much in release engineering? We ship a release every six weeks and a beta every week for our Firefox for Android and Firefox for desktop products. (Firefox OS is on a different cadence) The cost is too high to ship a bug to half a billion users. We want to make sure that when a patch is landed, the developer sees their test results in a reasonable amount of time. If there is a critical fix we need to get out to our users, we want to be able to ship a new release very quickly, usually within a day. If you can’t scale your infrastructure, you can’t scale the number of people you can hire.
And this is where we live.

We have two people in San Francisco, one in Vancouver, one in Thunder Bay, four in Toronto, two in Ottawa, one in Boston, one is Fairfax, VA, three in Dusseldorf, DE and one outside Christchurch New Zealand.

We are a very geographically distributed team and many of us work remotely. Even those people who work close to a physical Mozilla office such as Toronto or San Francisco work several days a week from home. Having such a distributed team is advantageous in that it allows us to hire the best release engineers around the world, not just those who live in Silicon Valley. Release engineering is a difficult skill to hire for. Also, it allows us to hand off work across timezones, such as when we are working on getting a release out the door.
Many people misunderstand the role of a release engineer within an organization. At Mozilla we provide operational support to the process that allows us to ship software. We are also tool developers. But instead of improving a product, we improve the process to ship.

A lot of people don’t like to think about process. I like this thought from Cate Huston, who is a mobile developer at Google.

“good process is invisible, because good process gets called culture, instead”

So now you know three things: many builds and tests, lots of hardware and over a dozen release engineers. How do we scale that?

References

photo http://www.flickr.com/photos/freefoto/5982549938/sizes/z/

http://www.catehuston.com/blog/2014/02/19/process-and-culture/
Here are some of projects that we use in our infrastructure.

Buildbot is our continuous integration engine. It’s an open source project written in Python. We spend a lot of time writing Python to extend and customize it.

We use Puppet for configuration management all our Buildbot masters, and the Linux and Mac slaves. So when we provision new hardware, we just boot the device and it puppetizes based on it’s role that’s defined by it’s hostname.

Our repository of record is hg.mozilla.org but developers also commit to git repos and these commits are transferred to the hg repository. We also use a lot of mozilla tools that allow us to scale. These tools are open source as well and I have links at the end of the talk to their github repos.

References
octokitty http://www.flickr.com/photos/tachikoma/2760470578/sizes/l/
We have many different branches in Hg at Mozilla. Developers push to different branches depending on their purpose. Within Buildbot we configure various branches to different scheduling priorities. So for instance, if a change is landed in a mozilla-beta branch, the builds and tests associated with that change will have machines allocated to them with a higher priority than if a change was landed on a cedar branch which is just for testing purposes.

References

branch picture http://www.flickr.com/photos/weijie1996/4936732364/sizes/l/
Try is a special branch that we have where developers can land their changes and ensure that the test results are green before these changes are landed on a production branch. This makes for less disruptions on production branches. This graph shows that the distribution of pushes for last January, you can see that there are around 50% on try.

Mozilla-inbound is an integration branch that merges to and from mozilla-central about once a day. It's a place where changes can land and be tested without risk of breaking the main mozilla-central trunk. Again, not a high priority branch. This branch is where a lot of community contributions come in.

Developers can also rent a project branch from release engineering and use this for testing with their team.

References
Tree rules re inbound https://wiki.mozilla.org/Tree_Rules/Inbound
This is tbpl. You don’t need to remember the name. It’s just a web page displays the state of build and test jobs to developers. It also allows you to cancel or retrigger them. So we have a lot of tools in the hands of developers to manage their builds and tests, instead of having to interact with a release engineer.

One of my coworkers noted that this screen looks like someone dropped a box of skittles on the floor, this is not inaccurate. You can see there are mostly green results but also orange (test failures) and blue (tests retried). Each line shows a build for that platform and all the tests that are running in parallel for that build on various slaves.
This is a picture of how the different parts of our build farm work together. Developers land change on code repositories such as hg.mozilla.org.

As I mentioned before, we use an open source continuous integration engine called Buildbot. We have over 50 buildbot masters. Masters are segregated by function to run tests, builds, scheduling, and try. Test and build masters are further divided by function so we can limit the type of jobs they run and the types of slaves they serve. For instance, a master may have Windows build slaves allocated to it. Or Android test slaves. This makes the masters more efficient because you don’t need to have every type of job loaded and consuming memory. It also makes maintenance more efficient in that you can bring down for example, Android test masters for maintenance without having to touch other platforms.

Buildbot polls the hg push log for each of the code repositories. (Hgpoller)

When the poller detects a change, the information about the change is written into the scheduler database. The buildbot scheduler masters are responsible for taking this request in the database and creating a new build request. The build request then will appear as pending in the web page in the previous slide.

The jobs may be on existing hardware in our data centre, or new VMs may start or be created in the cloud to run these pending jobs.
This is quite complicated environment and things can go wrong. When you land a change that causes a lot of red we call this burning the tree.

Everyone has an experience where their change burns a tree, and they have to back it out. It’s not a good experience, but we have process to mitigate the effects.

___

Reference

http://www.flickr.com/photos/ervins_strauhanis/9554405492/sizes/l/
http://armenzg.blogspot.ca/search?updated-max=2014-02-27T14:07:00-05:00&max-results=3
To deal with burning trees, you would think that we’d have firefighters. Instead, we have a team called sheriffs.

Their role is to reduce chaos. They watch the various branches of the tree and ensure that a change isn’t causing jobs to burn. There are about five sheriffs and they live in different timezones so that there’s global coverage. Some sheriffs are paid employees, some are volunteers. They back out change sets that cause problems and notify the offending party via bugzilla or irc. Or if there have been a lot of changes since a green build ran on that branch they will back out everything to the previous good state.

If someone releases a change that breaks the jobs on a branch, or if there is a network or infrastructure issue that causes jobs to burn, the sheriffs can close the branch. This means that no more jobs will run on that branch until the issue is resolved. This saves infrastructure money and people time.

Reference
Woody the sherriff picture http://www.flickr.com/photos/bakershut_/4736265729/sizes/l/
Release engineering also developed slave health tools that allow you to look at the status of various machines and their history. It used to be that sheriffs had to contact a release engineer to reboot a slave that was behaving badly but this is not very efficient. We like to put tools in the hands of people who need them. So you can see on this page that you can reboot a slave, look at its job history, or disable it from production.
So this infrastructure is all very complicated. How do we deploy changes to it without breaking the world?

As release engineers, we all have a staging Buildbot master environment where we can test our release engineering patches and ensure they will work in a production environment. We can also assign production devices to ourselves and invoke jobs from production change sets on our staging environment to replicate the real thing.

Code review is part of our culture. We use the bugzilla review flags for code review. We also use the feedback flag if we are starting something need for feedback from multiple teams.
No patch is landed without code review.
Every code change is checked against a pre-production continuous environment and we are notified in IRC if it fails.
We have to fix it or back it out before it is deployed to the production masters.

Reference
train image http://www.flickr.com/photos/wwworks/2942950081/sizes/l/
Another way we reduce chaos is to only deploy new code to our production environment at fixed intervals by the person on buildduty.

Buildduty is a rotating weekly role assigned to members of our team. This person serves as the deflector shield for the team from all developer requests. For instance, if a developer needs a slave loaned to them to try to debug a problem. Or if a certain class of machines are burning builds, they will investigate. They ensure we have enough build and slave capacity up to run the volume of jobs. The buildduty person is also responsible for running reconfigs a few times a week, which is a process to deploy all the code changes to the buildbot masters and make sure nothing unexpected happens. Stability. We use a command line tool called Fabric to deploy code to all of our 50 masters in parallel.

We also have a rotating role called release duty. These people are assigned to work with release management and QA to ensure our release automation works correctly for a given beta or release. This role can travel across timezones in a day, depending on how things go. Starting a release is as easy as entering a few fields in a web page.

As a release engineer you’ve been on a team where you work on tools, get interrupted to fix a build, work on tools, get interrupted to fix a machine and so on. Not very productive.
This is a street in Bangkok. As you can see, lots of traffic, not much movement. There used to be a problem at Mozilla where some platforms did not have good wait times because we simply didn’t have the slave capacity to handle them. Many pending tests. This was a source of frustration for developers.

We used to run all our builds and tests on in-house hardware in our data centres. This was inefficient in that it took a long time to acquire, rack and install the machines and burn them in. Also, we could not dynamically bring machines up to deal with peak load, and then put them offline when they were no longer needed.
So in early 2012 we started investigating how we could better scale this traffic.

We investigated running jobs on AWS starting with CentOS machines. One of the things that allowed us to move to AWS more easily was that we use Puppet to manage the configuration many of our build and test slaves (exception is Windows and Android). Our puppet modules are role based so the modifications required to add Amazon VMs were not that difficult.

This move to AWS provided additional capacity, and some of the machines in our data centres were repurposed to pools that were lacking capacity.

Reference

Cloud picture http://www.flickr.com/photos/paul-vallejo/2359829594/sizes/l/
Tests and builds in cloud = better wait times
Better wait times = happier developers
Initial success allowed us to continue this experiment because we had management support to fund this transition.

Reference
Mozilla Toronto summit
http://to.mozillacanada.org/mozilla-summit-2013-toronto/
And happier developers mean happy release engineers because we aren’t having to deflect as many problems with “why are the builds slow”.

So we moved more classes of slaves to AWS and the wait times got better again. It was awesome. But it was also getting expensive. Until recently, our AWS bill was 100K a month. So we needed to optimize this.
Optimizing our AWS use

- Run instances in multiple regions
- Start instances in cheaper regions first
- Scripts to automatically shut down inactive instances
- Start instances that have been recently running
- Use spot instances vs on demand instances for tests

We also use machines in multiple AWS regions, in case one region went down, and also to incur cost savings (some regions are cheaper)

Better build times and lower costs if you start instances that have recently been running (still retain artifact dirs, billing advantages)

We started out using Amazon’s on-demand instances. They cost more than other instance types and are available immediately for use. We recently switched some jobs to spot instances which are sold via auction by Amazon to sell off their unused capacity. So basically we are bidding for these instances, which are not available as quickly but are cheaper. Also, they can be killed at anytime if someone bids a higher price for them, but this didn’t happen that often (1%) and we have retry capabilities for our jobs within Buildbot. Spot instances turned out to work great for tests but not so well for builds. Our build machines are not wiped clean every time they run a build, they run incrementally. So when using spot instances we had to repuppetize every time from a blank slate which is not ideal for fast build times.
- So we have more capacity, but this revealed some more issues
- As I mentioned, we run incremental builds. This means that if a machine has recently run a certain build type, it will run faster the next time it runs that same type because it will just update the existing files on the machine, such as checkouts and object directories.
- One of the problems we have is that we have large pools of devices allocated to certain types of builds. This means that a build might not run on a machine that has recently ran a build of the same type. So a couple of people on the team looked at enabling smaller pools of build machines for certain types of builds. The nickname for these smaller types of build pools is a jacuzzi.
- Given a smaller pool, there would be a higher chance that the previous artifacts remain the next time the job ran.
- We use a tool called mock that installs packages in a virtual environments. They also optimized mock environments so that packages weren’t reinstalled if they already existed.
- These changes improved build times on these machines by 50%.
- The changes to use jacuzzis and spot instances reduced our AWS bill to around $75K a month
- Since the spot instances aren’t available as quickly as the on-demand instances, some tests don’t start within 15 minutes but that’s okay
- We implemented more jacuzzis this month and we are on track to have our Amazon bill reduced significantly again

Reference
So we’ve talked about optimizing using AWS. Another part of the puzzle is optimizing mobile. These are the platforms we test for Android.

Android testing

- Armv6
  - Android 2.2 - 200 Tegra devices in a lab - EOL soon
- Armv7
  - Android 4.0 on rack mounted reference devices
  - Android 2.3 - emulator in AWS
- x86
  - Android 4.2 - parallelized emulators
Most companies that do a lot of mobile device testing just have a roomful of devices that developers can test on.

We actually run continuous integration tests on Android reference cards. We have about 1000 of them in use. Two hundred of them are precariously sitting in a lab and are very finicky. I won’t talk about them because they are going away soon. The remaining 800 are called pandas and are rack mounted. These devices are not as stable as desktop devices, and are prone to failure. Given their numbers, having to deal with the machines failing all the time is very expensive if they were managed by humans.

In fact when we first had these rack mounted devices in production, they were rebooting spuriously all the time. ___

References
Pictures of Panda chassis from Dustin’s blog
https://blog.mozilla.org/it/2013/01/04/mozpool/2012-11-09-08-30-03/
We have a team at Mozilla called the Ateam. They are responsible for the automation tooling such as the test harnesses that run performance and unit tests. Anyways, they along with the IT group spent a lot of time trying to figure out why these devices were rebooting all the time - was the temperature in the chassis too high, was the load on the cpu too high, or was there a power issue. After months of debugging they found that the chassis had inadequate power and IT installed new power supplies.
Optimizing mobile

• Mozpool - reliably manage the imaging, rebooting, and verifying of mobile devices.

• Mozharness - scripting libraries to run common build and test oriented tasks i.e. fetch a zip, parse logs etc.

So aside from the hardware tweaks, the IT and release engineering teams wrote a tool called mozpool to manage these devices in an effective way. It has a simple REST API via HTTP. Mozclient libraries used by Mozharness to acquire devices to run tests. Mozharness used on other platforms too, not just mobile
Conclusions
Take small steps for improvement, implement, re-evaluate, reiterate
Every time you optimize something you’ll find another bottleneck
Invest in people writing tooling vs people doing repetitive tasks
Put tools in the hands of developers so they have control and we don’t need to interact with them unless there’s a serious issue
Code review and staged deployment
Commitment to funding release engineering to make the organization a whole more effective
Frame release engineering in terms of business goals to acquire funding - i.e. we want to be able to release every six weeks, and do a security release within 24 hours. Or if we want to hire 50 more developers to work on a new product, how do we do this we can’t scale our current load to include more capacity?

Reference
stairs photo http://www.flickr.com/photos/svenwerk/698875341/sizes/l/
Learn more

- @MozRelEng
- http://planet.mozilla.org/releng/
- IRC: channel #releng on moznet

To learn more about what we’re doing in the future
Further references

- Cloud tools: http://hg.mozilla.org/build/cloud-tools/
- Custom buildbot https://github.com/mozilla/build-buildbot-configs
- Mozharness https://github.com/mozilla/build-mozharness
- Mozpool https://github.com/mozilla/mozpool
- Puppet configs https://github.com/mozilla/build-puppet
- Git and hg bidirectional commits http://dtor.com/halfire/2013/05/20/2013_Releng_talk.html
Evaluate This Session

1. Sign-in: [www.eclipsecon.org](http://www.eclipsecon.org)

2. Select session from schedule

3. Evaluate: +1 0 -1