AWS testing results

Target: XXXXX

XXXXXX		
Memory	2 gig	
cpu	Dual Core: Intel(R) Xeon(R) CPU E5645 @ 2.40GHz	

Testing Procedure

This is somewhat complicated, and a lot of steps, but it all fits together

Prepare Target Machine

Prepare modified mosquitto configuration

The target needs to have mosquitto installed and configured as per as desired system configuration. The easiest/most correct way of doing this is to deploy APP - Keypit via it's regular fab file. That installs the correct version, installs ACLs and the correct security configuration. However, we will make one modification as we want to install a different keyfile (The big list of all TLS-PSK keypairs that our testing machines will use)

Locally, in your https://github.com/remakeelectric/mqtt-malaria tree, run

```
./malaria keygen -n 20000 > malaria.pskfile
scp malaria.pskfile target.machine:/etc/mosquitto
# Edit /etc/mosquitto/mosquitto.conf to
```

Edit /etc/mosquitto/mosquitto.conf as follows to use our testing keyfile

/etc/mosquitto/mosquitto.conf

```
#psk_file /etc/mosquitto/keypit.pskfile
psk_file /etc/mosquitto/malaria.pskfile
```

You only need to do this once for the entire testing period.

Install Anecdotal monitoring tools

For anecdotal monitoring, I used htop, but you could also just use top, or dstat or whatever you prefer to look at.

```
apt-get install htop
htop
```

Install and run MQTT monitoring (malaria)

The malaria project contains a fabfile that helps deploy malaria to any target you like. This installs malaria into a virtualenv in /tmp

```
# In your malaria virtual env
fab -H target.machine cleanup deploy
```

Now, start the less intrusive watching process (As opposed to malaria's time tracking and duplicate checking modes)

```
karlp@yourmachine$ ssh target.machine
karlp@target.machine$ cd /tmp
karlp@target.machine$ mkdir /tmp/mqttfs
karlp@target-machine$ . $(cat malaria-tmp-homedir)/venv/bin/activate
(venv)karlp@target.machine:/tmp/malaria-tmp-XXXX$ malaria watch -t "#" -d /tmp/mqttfs
```

You'll have to leave this running in one SSH console, it doesn't have support for starting and running as a service (yet)

Install and run system monitoring/graphing support (vmstatplot)

The vmstatplot project contains a fabfile for deploying as a service, but you may need to modify the fabfile to tweak the paths.

```
fab -H target.machine start
```

You probably want to ssh to the machine afterwards and check that it's running:

```
status vmstatplot
# Potentially: edit /etc/init/vmstatplot.conf
# Potentially: sudo start vmstatplot
tail -f /tmp/vmstatplot/output.log
```

Wait 2-3 minutes, and then check that you have a viable baseline. (This collects the last 500 lines of the log file)

```
fab -H target.machine collect:500
```

Prepare Attack

You can use "malaria publish" directly from your own machine, you can use vagrant VMs, or you can use AWS t1.micro instances as attack nodes. Here we will use AWS t1.micros

Prepare AWS credentials

Make a ~/.boto file like so

~/.boto

```
[Credentials]
aws_access_key_id = GET_THESE_FROM_
aws_secret_access_key = _YOUR_AWS_CONSOLE_WEBPAGE
```

If you don't know the secret part, you'll need to make a new credential, but that's something for you to work out!

Start some bees

In your mqtt-malaria virtualenv

```
$ fab beeup:2  # just start two bees, until you know what you are doing!
('Reservation is ', Reservation:r-47cc790b)
Waiting for Amazon to breed bees
.
Bee i-8ae89ac6 ready for action!
Bee i-8be89ac7 ready for action!
Adding ubuntu@ec2-54-229-239-145.eu-west-1.compute.amazonaws.com to our list of
workers
Adding ubuntu@ec2-54-229-239-147.eu-west-1.compute.amazonaws.com to our list of
workers
Done.
```

Install attack software (malaria)

You need to do a couple of steps here, that are broken apart as not all machines need them in all orders  (It's a bit different with brand new AWS machines vs restored vagrant machines and so on)

```
# load machine definitions from malaria state, and run apt-get update
fab -i /path/to/AWS/ssh/private-key.file.pem mstate aptup
#.....
fab -i /path/to/AWS/ssh/private-key.file.pem mstate deploy:True
# The "True" argument installs mosquitto as well, for a general testing case, you may
have
# preinstalled/configured whatever mosquitto you wanted to test with, but we want
latest in this case
# .....
```

The deploy stage will take a while the first time, as it needs to install a bunch of python dev libraries, and it's not fully parallel yet.

Share keys

If you're planning on running any SSL tests, you probably need to share out some keys. Remember the keyfile we generated earlier? Now we're going to chunk that out to all the attack nodes. This will split the keyfile evenly among all nodes in the swam.

```
fab -i ~/.ssh/karl-malaria-bees-2013-oct-15.pem mstate share_key:malaria.pskfile
```

Run tests

This is more open ended, you need to plan your warheads in conjunction with how many attack nodes you have so you get the message rates and volumes that you expect

Run attack (from mqtt-malaria virtualenv)

```
fab -i ~/.ssh/karl-malaria-bees-2013-oct-15.pem mstate
attack:mqtt.example.com,warhead=warheads/20x5mps--1000x500bytes.wh
```

in vmstatplot virtualenv

```
fab -H target.machine collect:500
```

Post Testing Cleanup

Turn off all attack bees

In your mqtt-malaria project virtualenv

```
fab down
```

You may want to double check in your AWS console that they are all terminated 🙂

Turn off and remove graphing/syslogging collections

```
fab -H target.machine stop
```

Turn off and remove MQTT monitoring

```
# CTRL-C the mqttfs watching process you ran earlier
rm -rf /tmp/malaria-tmp-*
```

Restore mosquitto config

- `Restore the psk_file configuration you edited earlier`
- Remove the malaria.pskfile itself

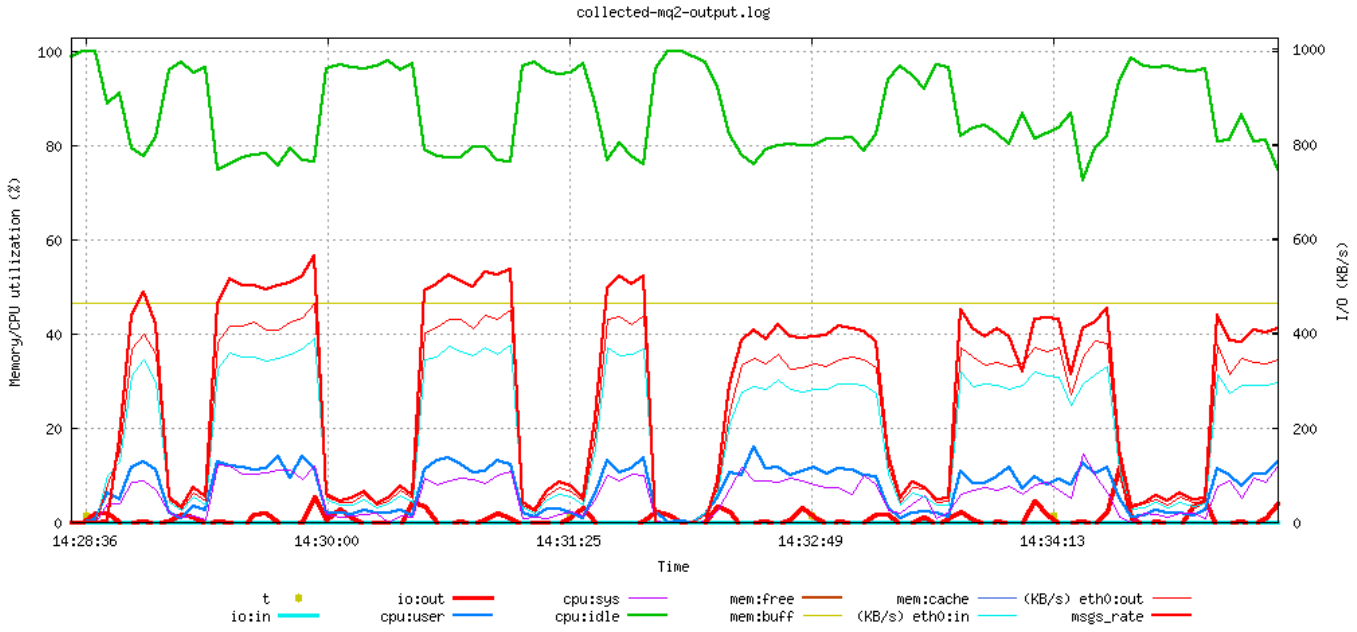
Tips/Tricks/Lessons

A t1.micro instance running malaria publish -P 100 will run about 55-60% cpu

200 will run, mostly, (after tweaking the delays to allow files to flush and mosquitto brokers to start up) but it runs very rough, at 95% cpu and 95% mem, 150 processes is probably a more sane maximum per t1.micro instance. The message throughput flucuates wildly from 250 up to almost 1000, (which would be the ideal rate of 200 procs at 5 msgs/sec) It would even have periods of only 20-40 msgs/sec. No drops occur, we're simply overloading the linux kernel and it's scheduling on the little t1.micro (load average reached 260)

In fact, even with only 100 processes, it simply can't sustain 500msgs per sec, it will occasionally churn and drop down to 30-40 per second.

See below, first half is with 100clients at 5 msgs per sec, second is 100 clients at 4 msgs per sec, the drops are periodic....



100 clients at 3 msgs per second is better, but still hits churning eventually.

50 clients at 6 per second seems to be perfectly stable, but that's only 50 clients (even though the message rate is the same, that's a lot less ssl connections)

100 clients at 2 msgs per second seems rock solid (30% cpu, 307/595meg ram). 200 clients at 1 msgs per second however falls apart immediately, it's simply too much memory and cpu for the machine. 150 clients at 1.5 message per second *seems* solid too, about 220-230 msgs per sec on the target, 20% cpu, 380/595meg ram. (it has very short drops, just for a second or so, nothing as severe as the other failures)

200@0.5 works ok too it seems

Process Count	Max Rate	Notes
50	6 (300mps)	Might be more, this was fine
100	2 (200 mps)	
150	1.5 (225 mps)	
200	0.5 (100mps)	200@1 is too high, you *might* get more, but this is very close to ram limits
250	0.1	this is memory bound, 250 python instances just simply adds up! However, it works on a t1.micro!
300	0.1	this will CRASH! you will lose your ssh connection, and the device will crash and burn