“Hold this for a moment”

Real-world queuing strategies

David Dawson
Director of Technology

Marcus Kern
VP, Technology

velti

WHAT DO WE DO ?

BRAND BLOTTER

Our instantly available predictive analysis and personalisation tool provides a single view of your brand from all your dispersed data points and overlays sales data in real time so you can manage your mobile campaigns “in action”.

VELTI PLATFORMS

Powerful mobile technology that puts ideas in motion – an mCMS and a mobile campaign platform available for both: self-service or managed service.

VELTI MEDIA

The complete mobile advertising solution. Our own ad network & exchange, equipped with dynamic “real time” analytics of all your mobile activity using our Visualise tool, all under one roof.

MOBILE MARKETING

The complete mobile engagement solution. We help brands progress along their mobile roadmap, from fast growth pilots to optimisation of current assets and revenue growth.

MCRM

We build your mCRM engine that builds opt in customers into a mobile database and pushes it through the measurement tool so we can show you what you spend and what you gain.

LARGE SCALE CAMPAIGNS

Cultivate relationships that build excitement through fun and interesting experiences they want to participate in. From on-pack promos to premium competitions.

LOYALTY

Rewards based performance marketing, aimed at increasing customer lifetime value, revenue growth and acquisition of insightful consumer analytics. We provide both the programme and loyalty fulfillment.

BBC itv CHANNEL5 sky SONY vodafone MTV MUSIC TELEVISION O2

Velti Technologies

- Erlang
- RIAK & leveldb
- Redis
- Ubuntu

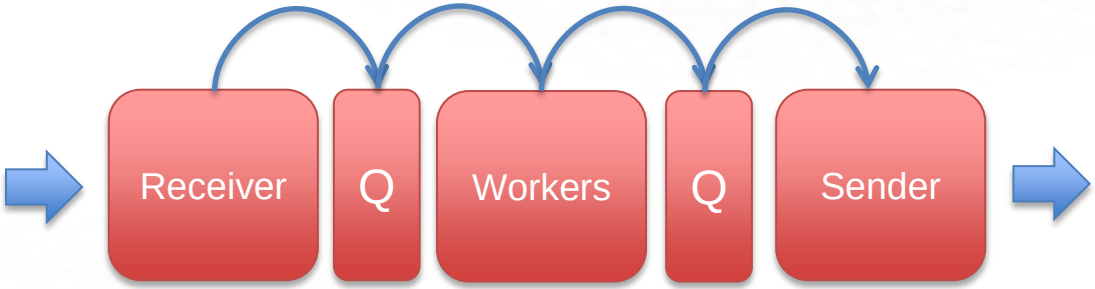
- Ruby on Rails
- Java
- Node.js
- MongoDB
- MySQL

Two parts to this story

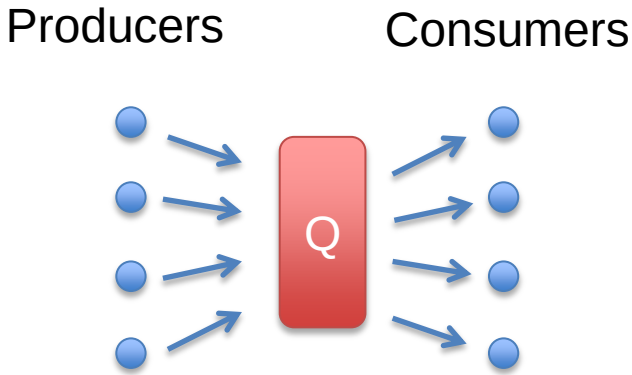
- Queuing Strategies

- Optimizing hardware

Building a Robust Queue

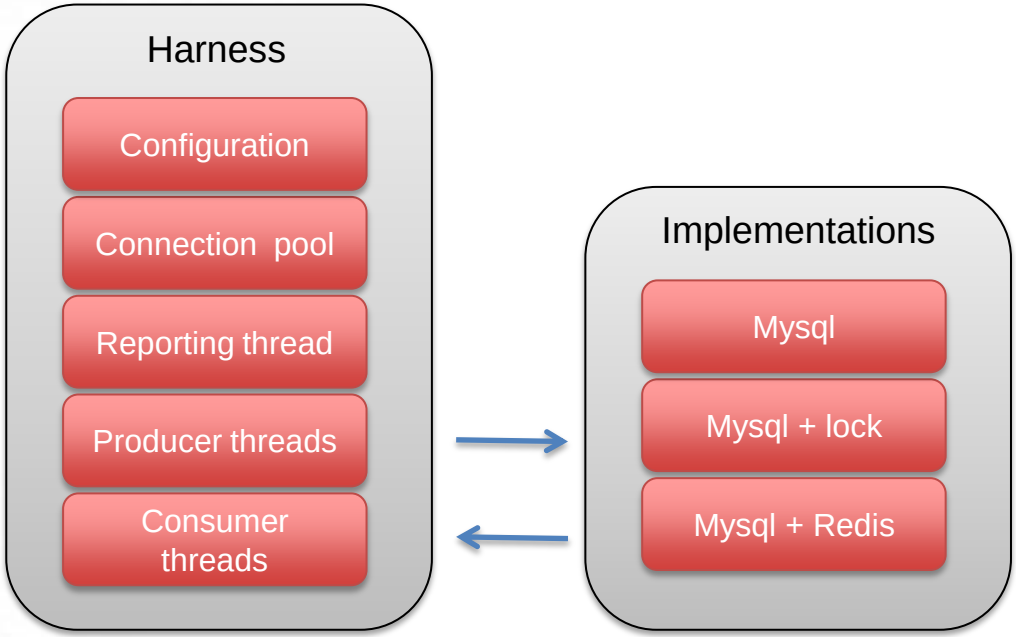


- Reliable + Replicated
- Scheduled jobs + Retries
- High performance (>10,000 tx/sec)
- Multiple producers and consumers (> 100)
- Easy to debug + Operations friendly



Test Harness

- Built using Jruby
  - Fast (Hotspot)
  - Threads without the GIL
- Pluggable design
  - Multiple implementations
- Configurable variables
  - Batch size
  - Number of Producers and Consumers
  - Number of itterations
- Reporting



Implementation #1

- Mysql only (v5.5) percona
- Innodb (xtradb)
- Replication
- 1 x table (‘queue’)
  - Id (primary key, auto_inc, int)
  - Worker_id (int)
  - Process_at (datetime)
  - Payload (varchar)
  - Index (worker_id, process_at)
- Dedicated hardware
  - Harness: HP DL365 (12 cores)
  - Mysql: HP DL365 (12 cores)

Implementation #1

velti

Multiple write operations

Insert into queue (worker_id, process_at, payload)
values (**0**, **'2012-01-01 01:01:00'**, **'{ json}'**)

Insert into queue (worker_id, process_at, payload)
values (**0**, **'2012-01-01 01:01:00'**, **'{ json}'**)

id	worker_id	process_at	payload
1	-1	2012-01-01 01:01:00	{ json }
2	-1	2012-01-01 01:01:00	{ json }
3	-1	2012-01-01 01:01:00	{ json }

Batched update / read operations

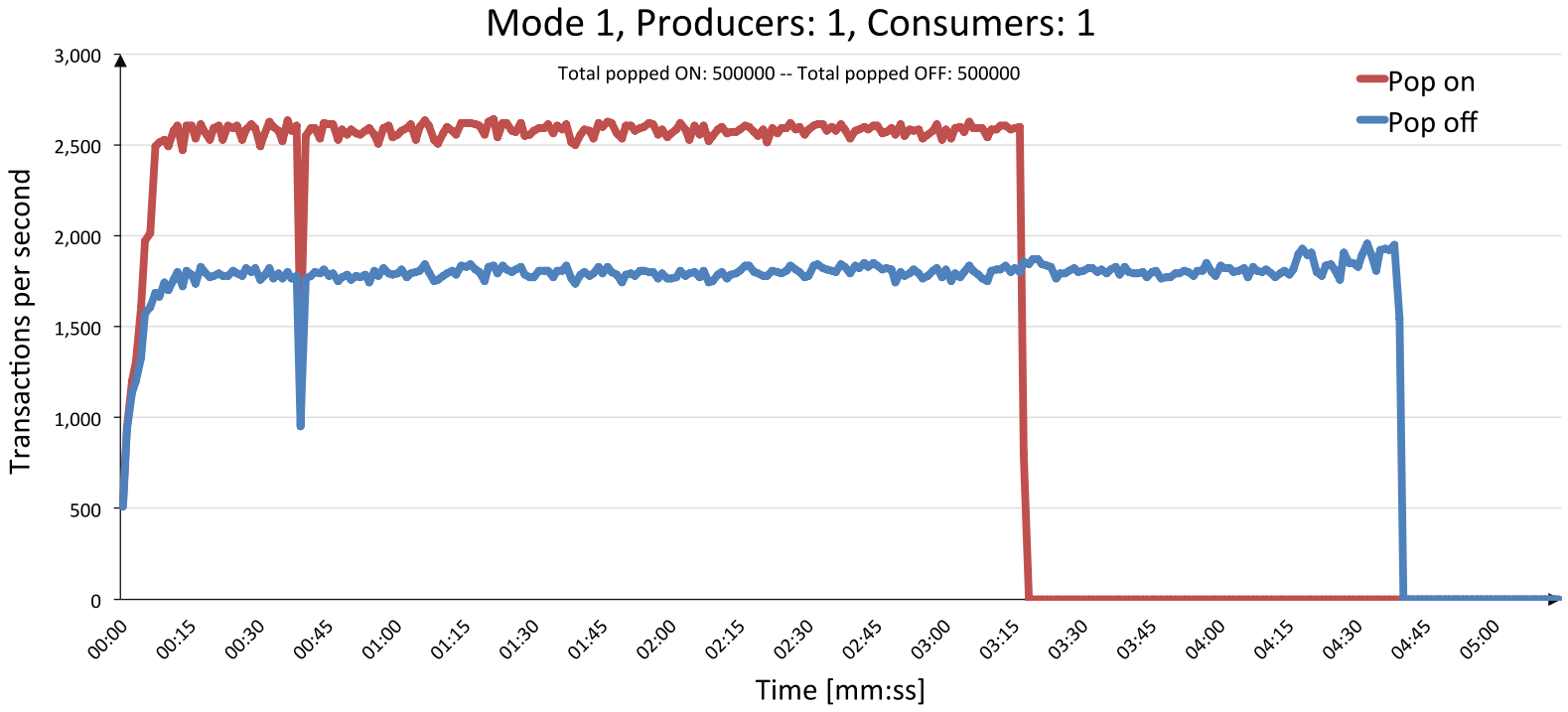
Update queue set worker_id=**123** where worker_id=**0** and process_at <= now() limit 10

Select * from queue where worker_id=**123**

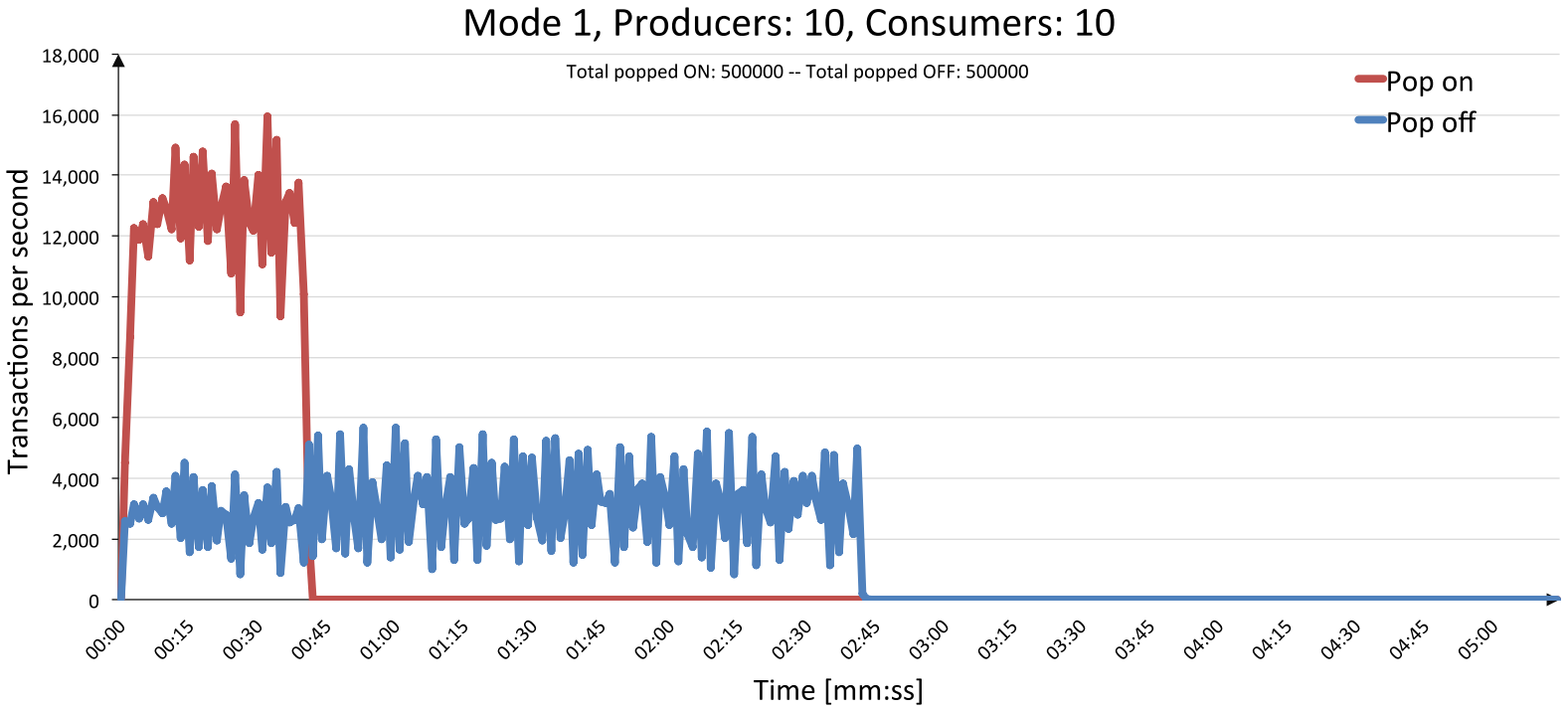
Update queue set worker_id=**-1** where id=**2**

Update queue set worker_id=**-1** where id=**3**

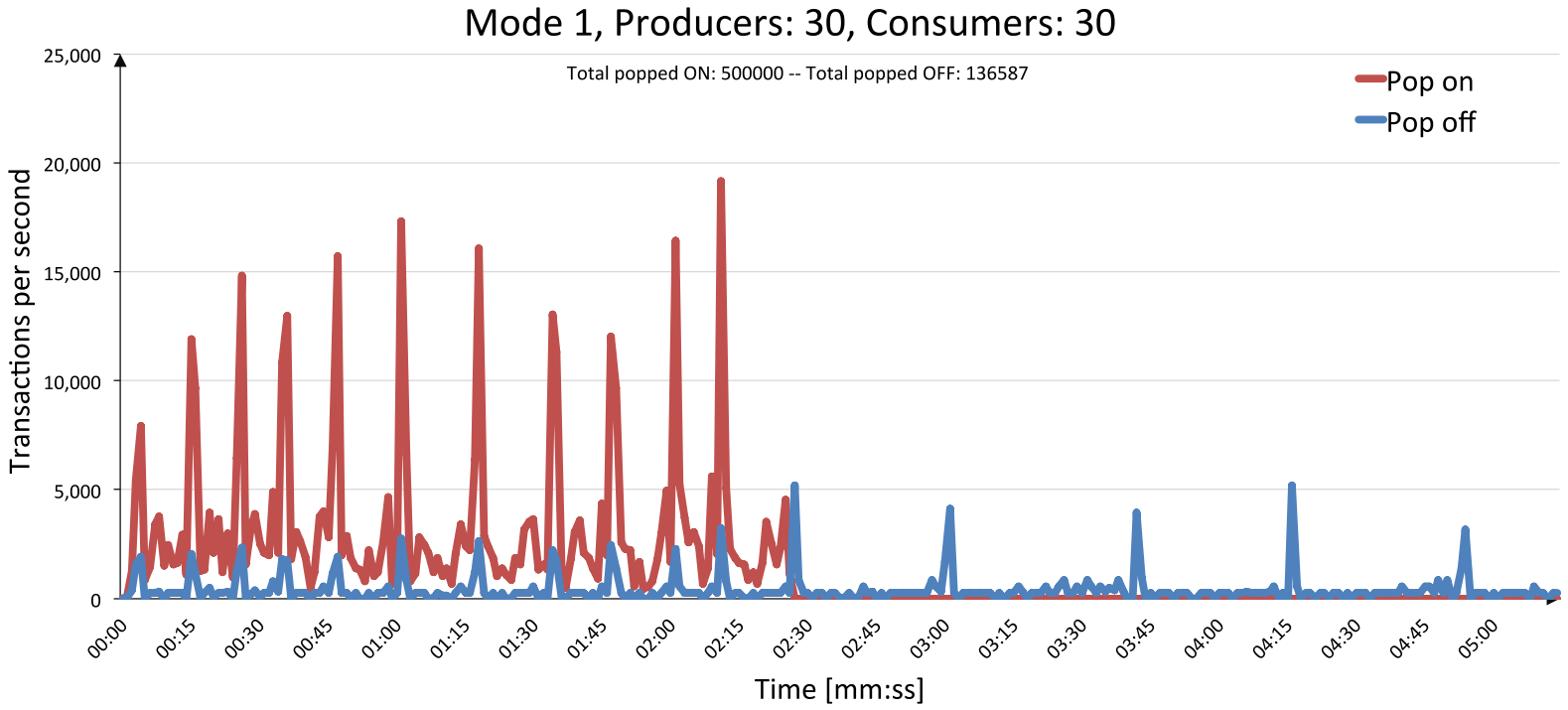
Implementation #1



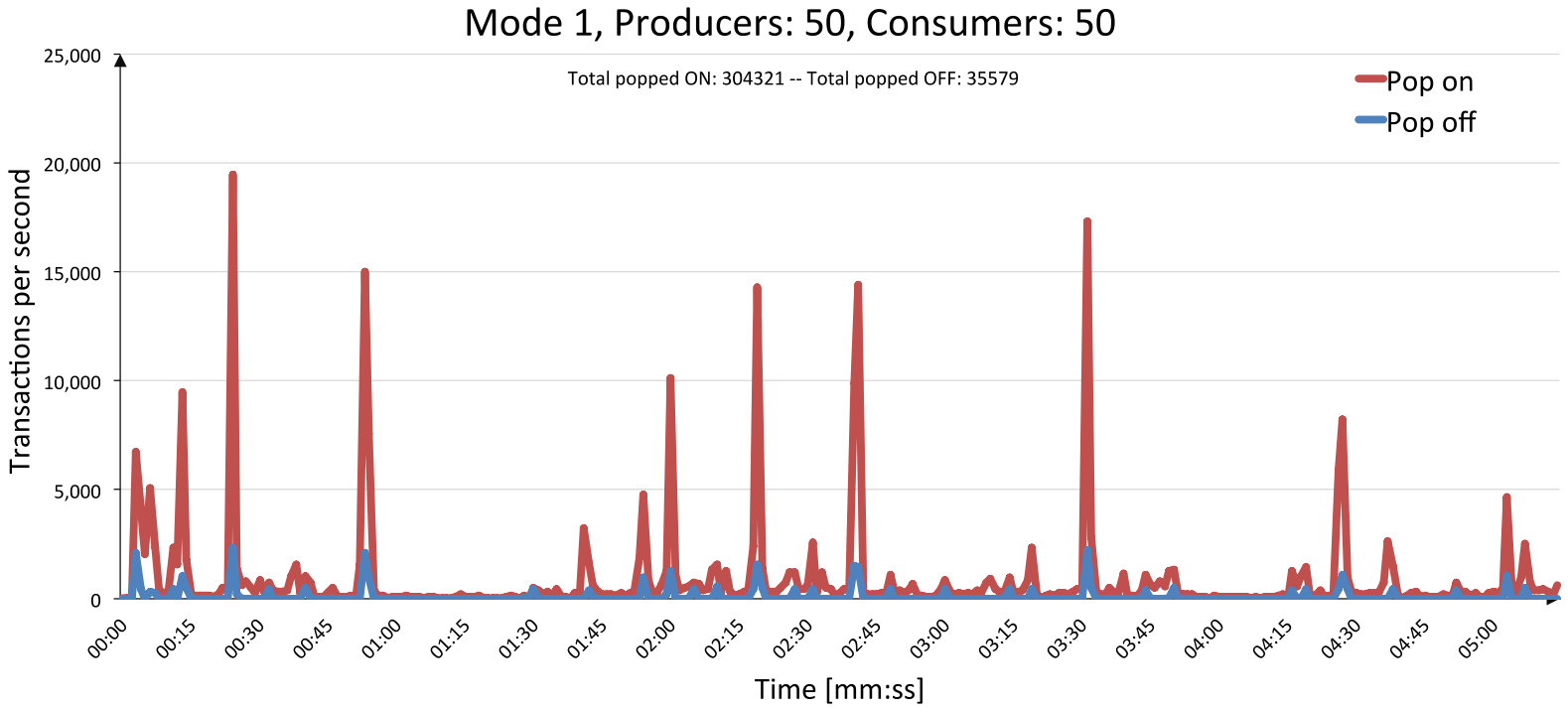
Implementation #1



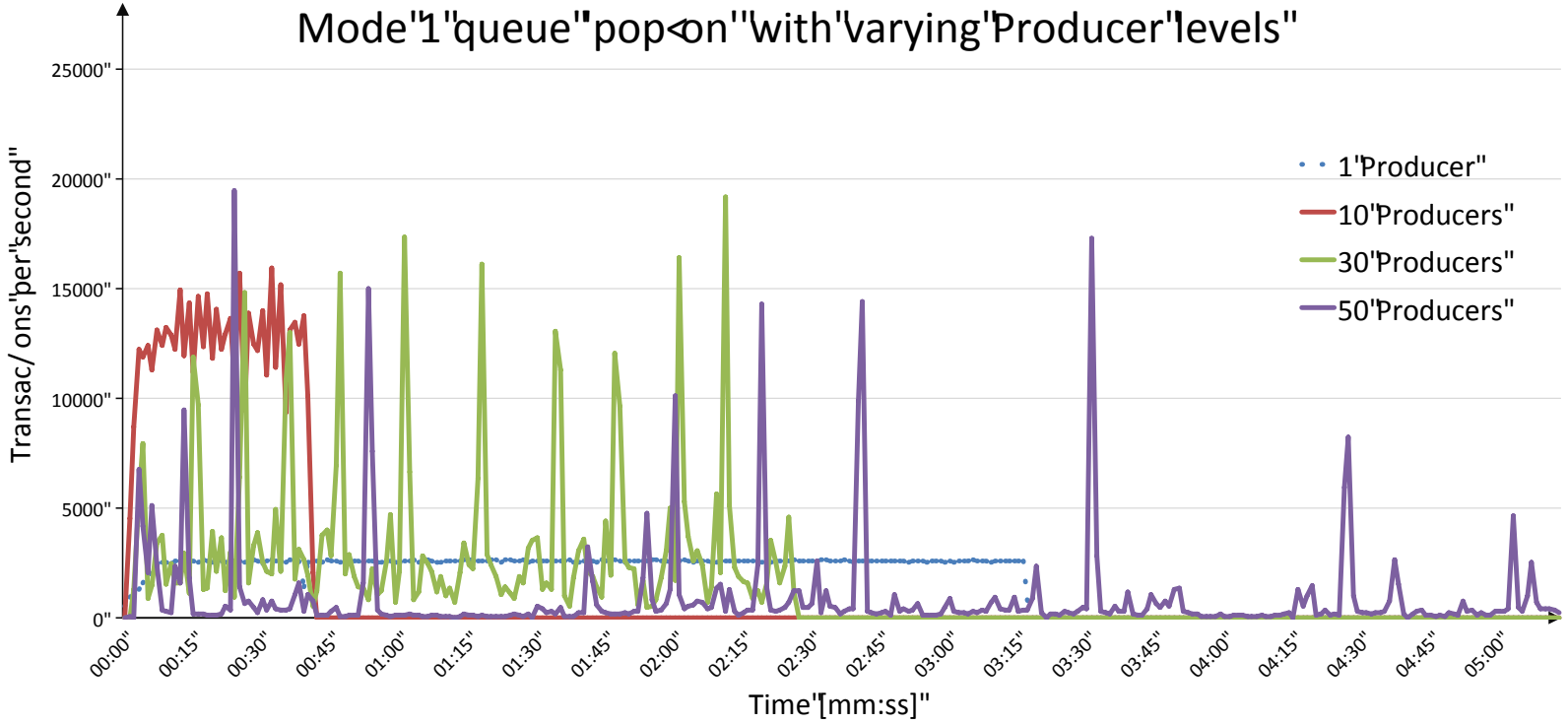
Implementation #1



Implementation #1



Implementation #1



Implementation #2

- Same Mysql setup as implementation #1
- Although we wrap a lock around the point of most contention (batch update)
  - Select get_lock(str, timeout)
  - Select release_lock(str)

Implementation #2 (mysql + Lock)

velti

Multiple write operations

Insert into queue (worker_id, process_at, payload)
values (**0**, **'2012-01-01 01:01:00'**, **'{ json}'**)

Insert into queue (worker_id, process_at, payload)
values (**0**, **'2012-01-01 01:01:00'**, **'{ json}'**)

id	worker_id	process_at	payload
1	-1	2012-01-01 01:01:00	{ json }
2	-1	2012-01-01 01:01:00	{ json }
3	-1	2012-01-01 01:01:00	{ json }

Batched update / read operations

Select get_lock('queue',-1)

Update queue set worker_id=**123** where worker_id=**0** and process_at > now() limit 10

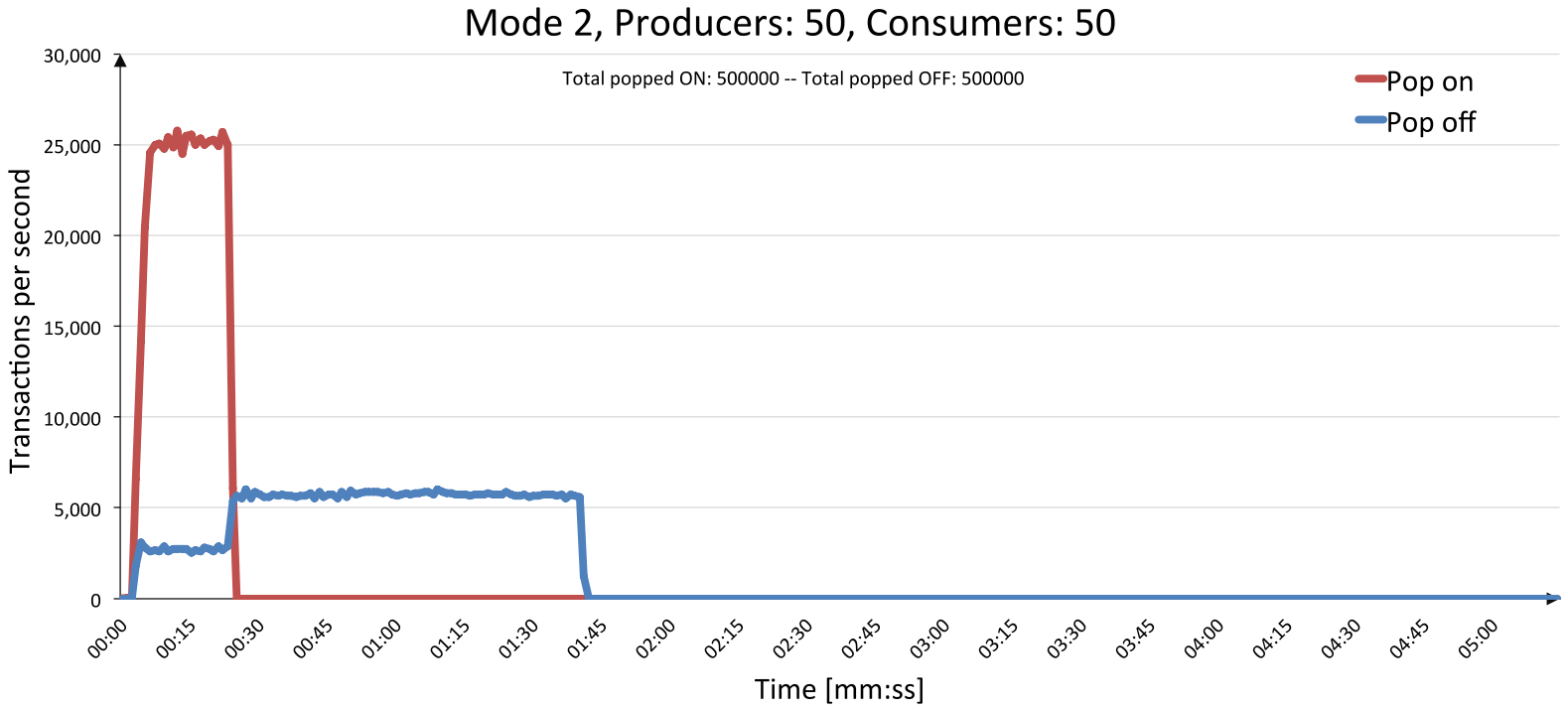
Select release_lock('queue')

Select * from queue where worker_id=**123**

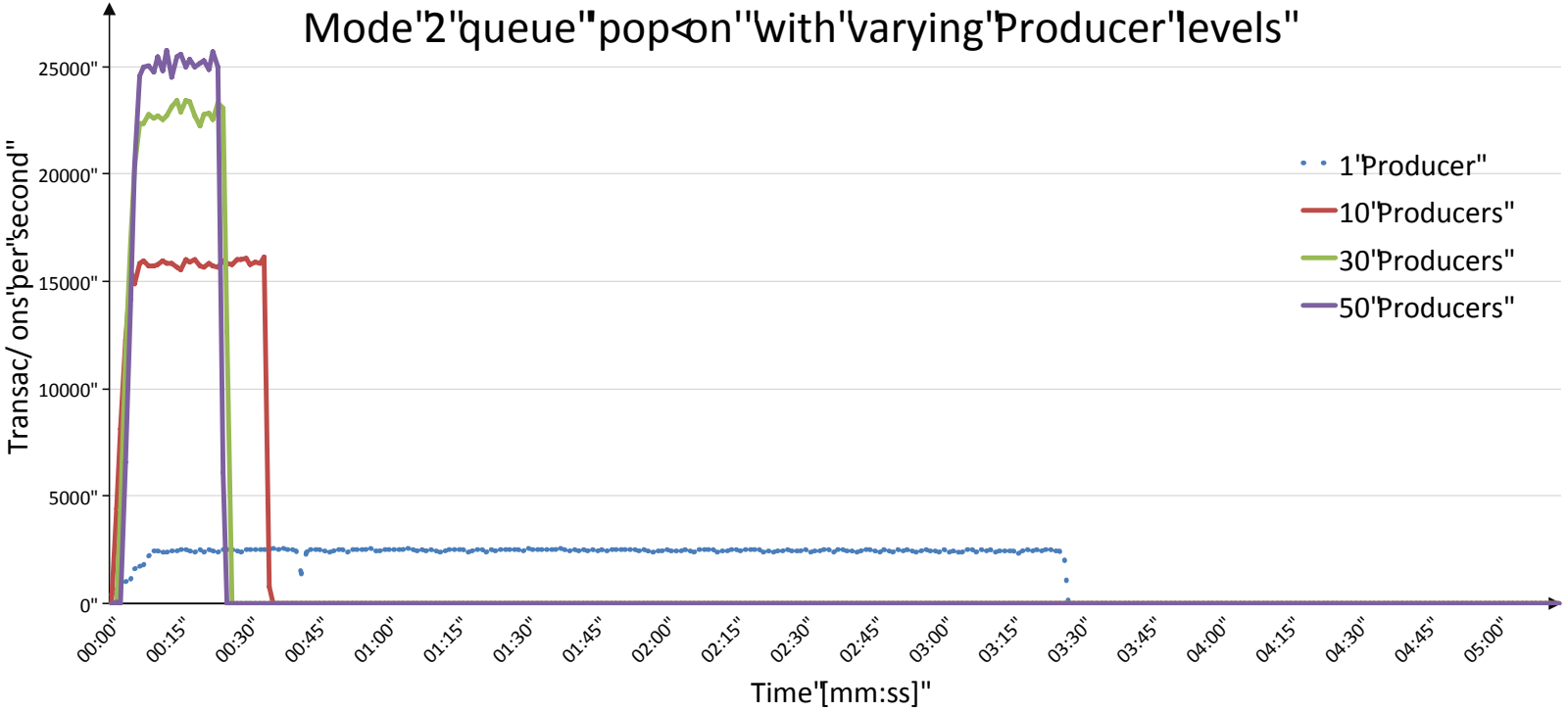
Update queue set worker_id=**-1** where id=**2**

Update queue set worker_id=**-1** where id=**3**

Implementation #2



Implementation #2



Implementation #3

- Same Mysql setup as implementation #1
- 1 x table (‘queue’)
  - Id (primary key, auto_inc, int)
  - Status(enum)
  - Process_at (datetime)
  - Payload (varchar)
- 1 x Redis using the following data structures
  - SortedSet (range query, schedule jobs)
  - Queue (fast push / pop sematics)
- Dedicated hardware
  - Harness: HP DL365 (12 cores)
  - Mysql + Redis: HP DL365 (12 cores)

Implementation #3

velti

Multiple write operations

Insert into queue (worker_id, process_at, payload)
values (**0**, **'2012-01-01 01:01:00'**, **'{ json}'**)

RedisQueue.push(**2, '2012-01-01 01:01:00'**)

Insert into queue (worker_id, process_at, payload)
values (**0**, **'2012-01-01 01:01:00'**, **'{ json}'**)

RedisQueue.push(**3, '2012-01-01 01:01:00'**)

Batched update / read operations

RedisQueue.pop(**'2012-01-01 01:01:00' , 10**)

Update queue set status='working' where id in (**2,3**)

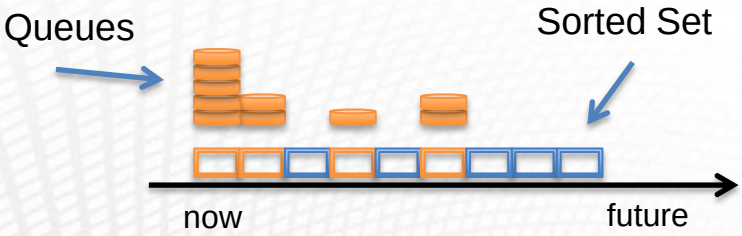
Update queue set status='finished' where id = **2**

Update queue set status='finished' where id = **2**

id	status	process_at	payload
1	'finished'	2012-01-01 01:01:00	{ json }
2	'finished'	2012-01-01 01:01:00	{ json }
3	'finished'	2012-01-01 01:01:00	{ json }

Implementation #3

- Redis Sorted Sets O(log N) complexity
  - Zadd/ zrangebyscore /zrem
  - Used to store the name of the queue and when it should be processed
- Redis Queues O(1) complexity
  - Rpush / lpop
  - User to store the items that need to be processed



RedisQueue.push

queue_name = ‘queue’ + **scheduled_time**

rpush(**queue_name**, **id_of_mysql_insert**)

zadd(‘q_set’, **scheduled_time**, **queue_name**)

RedisQueue.pop

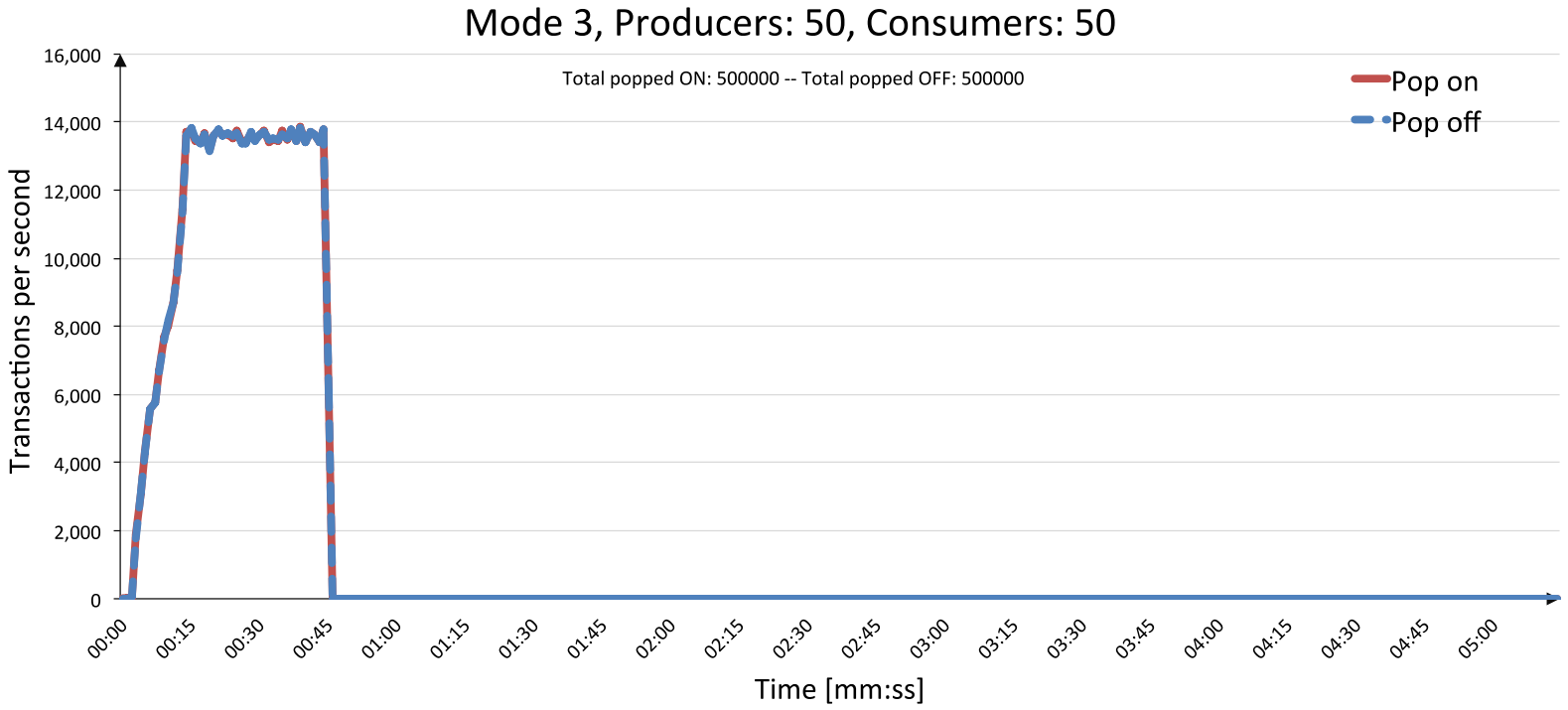
queue_name = redis.zrangebyscore('q_set', 0, **current_time**, :limit => [0,1])

Item = lpop(**queue_name**)

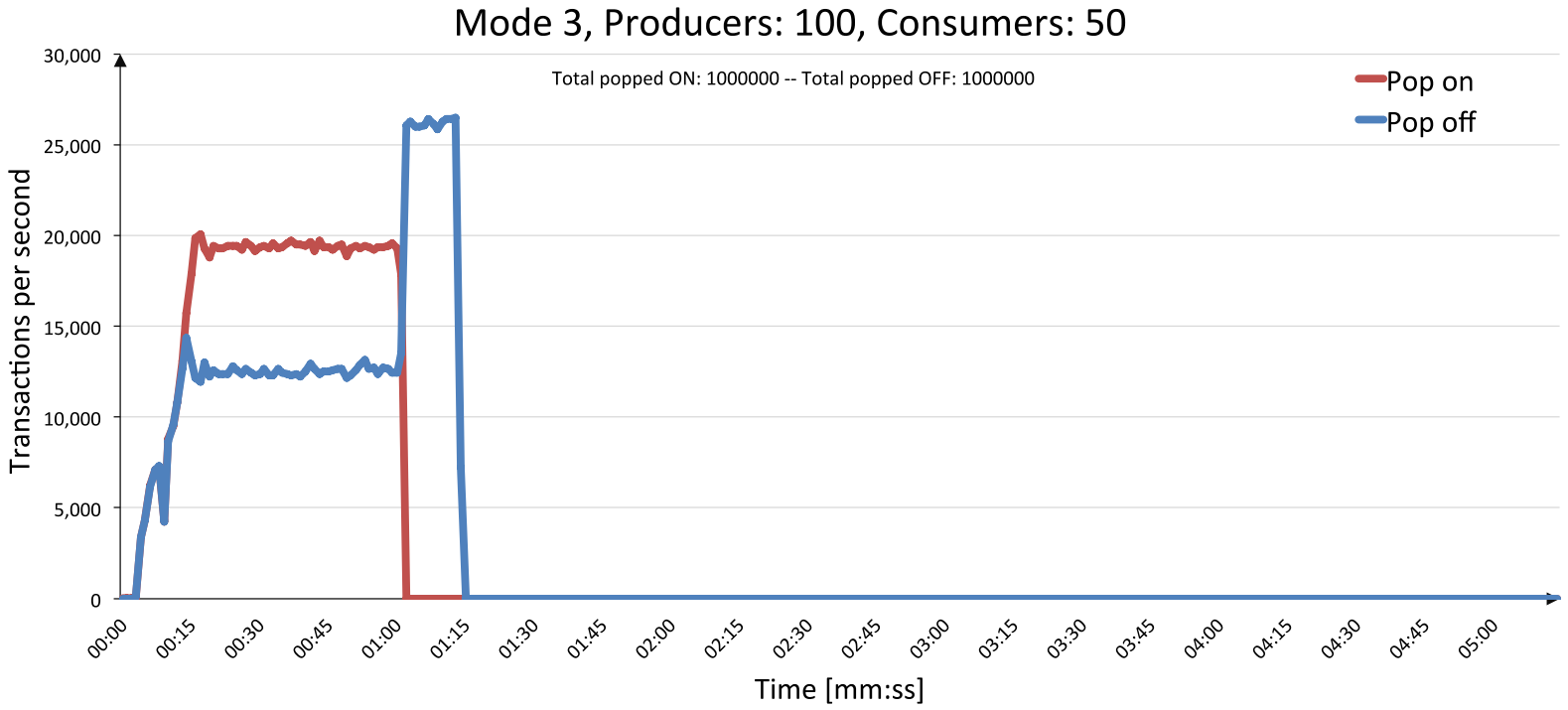
If item.nil?

Zrem(‘q_set’, **queue_name**)

Implementation #3



Implementation #3



Summarize Results

velti

Implementation #1

- Simplest Option ✓
- 1 Moving part ✓
- Easy to diagnose ✓
- Tried and tested ✓

- Prone to deadlocking ✗
- Contention ✗
- Slowest solution ✗

Implementation #2

- Less deadlocks ✓
- Easy to diagnose ✓
- Removed Contention ✓
- Big speed boost ✓

- Still deadlocks (rare) ✗
- Yet to be proven in production ✗

+

Summarize Results

velti

Implementation #3

- Fastest ✓
- No Contention ✓
- Predictable ✓
- Tried and tested ✓
- Dynamic queues ✓

- Most complicated ✗
- Recovery scripts ✗
- Multiple moving parts ✗

+ redis

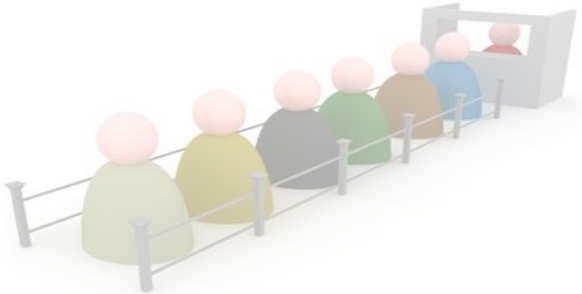
Future Considerations

- Currently limited by speed of Mysql
- Try a distributed key-value store
  - Recovery?
  - Eventual consistency?

riak + redis

Two parts to this story

- Queuing Strategies



- Optimizing hardware

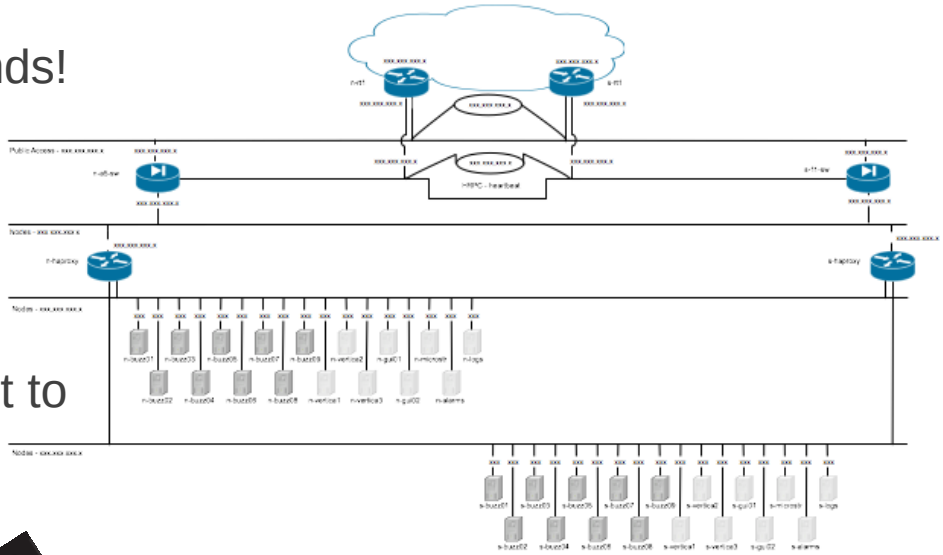
Hardware optimisation

- Observed **‘time outs’**
  App ⇔ RIAK DB
- Developed sophisticated balancing mechanisms to code around them, but they still occurred
- Especially under load

Photograph and Logo © 2010 Time Out Group Ltd.

Nature of the problem

- Delayed responses of up to 60 seconds!
- Our live environment contains:
  - 2 x 9 App & RIAK Nodes
  - HP DL385 G6
  - 2 x AMD Opteron 2431 (6 cores)
- We built a dedicated test environment to get to the bottom of this:
  - 3 x App & RIAK Nodes
  - 2 x Intel Xeon (8 cores)



Looking for contention…

Contention options

- CPU

Less than 60% utilisation

- Disk IO

- Got SSD (10x), Independent OEM
- RIAK (SSD) / Logs/OS (HDD)

- Network IO

- RIAK I/O hungry
- Use second NICs/RIAK VLAN

Memory contention / NUMA

- Looking at the 60% again
  - ***Non-Uniform Memory Access*** *(**NUMA**) is a computer memory design used in Multiprocessing, where the memory access time depends on the memory* ***location*** *relative to a processor. -* Wikipedia
- In the 1960s CPUs became faster then memory
- Race for larger cache memory & Cache algorithms
- Multi processors accessing the same memory leads to contention and significant performance impact
- Dedicate memory to processors/cores/threads
- BUT, - most memory data is required by more then one process. => cache coherent access (ccNUMA)
- Linux threading allocation is challenged
- Cache-coherence attracts significant overheads, especially for processes in quick succession!

Gain control! - NUMACTL

- Processor affinity – Bind a particular process type to a specific processor
- Instruct memory usage to use different memory banks
- For example: numactl --cpunodebind 1 –interleave all erl
- Get it here: apt-get install numactl

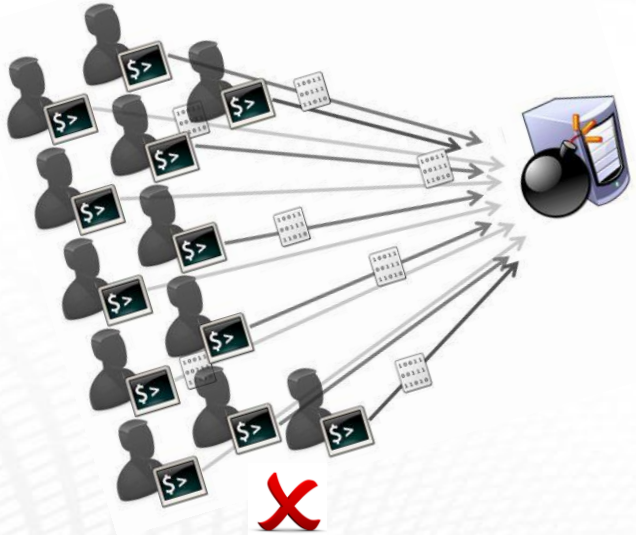
- => No timeouts
- => 20%+ speed increase when running App & RIAK
- => Full use of existing hardware

How about load testing ?

- Our interactive voting platform required load testing
- Requiring 10,000's connections / second
- Mixture of Http / Https
- Session based requests
  - Login a user
  - Get a list of candidates
  - Get the balance
  - Vote for a candidate if credit available

Load testing - lessons learned

velti



nn x AWS
⇒ Tsung SSL SessionID bug

ASA5520 limited at 3-4k new connections per second
⇒ Replaced with ASA5585 (Spec 50k/s, Tested 20k/s)

HAProxy on 2xDL120
⇒ # of Linux procs 1 -> 4
⇒ Added conn. Throttle 4k/server

6 x DL360 G6
⇒ Apache Cipher reduction
⇒ K/A consumed all threads -> reduced & disabled
⇒ Ulimit per proc 1k -> 65k

WAN ✓

FW ✗

LAN ✓

LB ✗

Servs ✗

Load testing Tools

- ab (apache bench)
  - Easy to use ✓
  - Lots of documentation ✓
  - Hard to distribute (although we did find “bees with machine guns”) ✗
    - https://github.com/newsapps/beeswithmachineguns)
  - We experienced Inconsistent results with our setup ✗
  - Struggled to create the complex sessions we required ✗

- httperf
  - Easy to use ✓
  - Lots of documentation ✓
  - Hard to distribute (no master / slave setup) ✗

Load testing Tools

- Write our own
  - Will do exactly what we want ✓
  - Time ✗
- Tsung
  - Very configurable ✓
  - Scalable ✓
  - Easier to distribute ✓
  - Already used in the department ✓
  - Steep learning curve ✗
  - Setting up a large cluster requires effort ✗

Tsung

- What is Tsung?
  - Open-source multi-protocol distributed load testing tool
  - Written in erlang
  - Can support multiple protocols: HTTP / SOAP / XMPP / etc.
  - Support for sessions
  - Master slave setup for distributed load testing

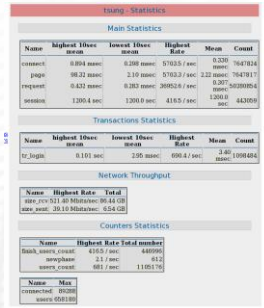
  - Very configurable ✓
  - Scalable ✓
  - Easier to distribute ✓
  - Already used in the department ✓
  - Steep learning curve ✗
  - Setting up a large cluster requires effort ✗

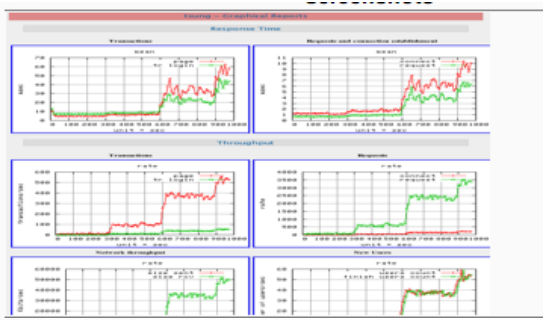
Distributed Tsung

- Although Tsung provided us almost everything we needed
- We still had to setup lots of instances manually
- This was time consuming / error prone
- We needed a tool to alleviate and automate this
- So we built……

Ion Storm

- Tool to setup a Tsung cluster on multiple EC2 instances
- With co-ordinated start stop functionality
- Written in ruby, using the rightscale gem:
  rightaws.rubyforge.org
- Which uploads the results to S3 after each run





Performance

- From a cluster of 20 machines we achieved
  - 20K HTTPS / Sec
  - 50K HTTP / Sec
  - 12K Session based request (mixture of api calls) / Sec

- Be warned though
  - Can be expensive to run through EC2
  - Limited to 20 EC2 instances unless you speak to Amazon nicely
  - Have a look at spot instances

Open Sourced!

- Designed and built by two Velti engineers
  - Ben Murphy

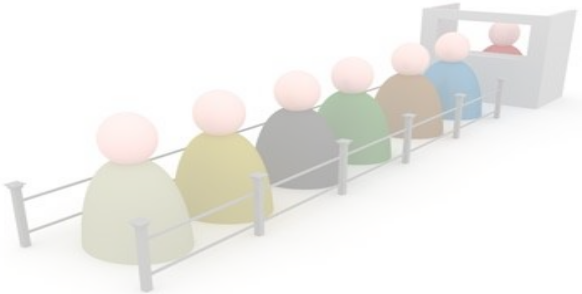
  - David Townsend

- Try it out:

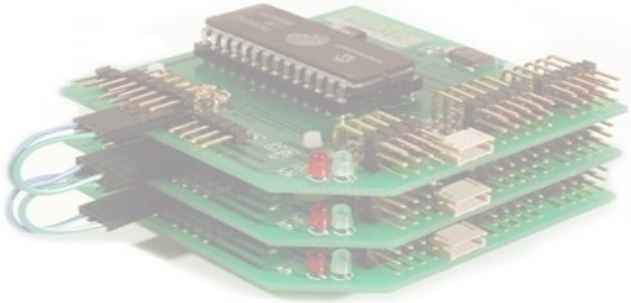
git@github.com:mitadmin/ionstorm.git

Two parts to this story

- Queuing Strategies



- Optimizing hardware



Thank You

Questions?

If you'd like to work *with* or *for* Velti please contact the Velti Team:

David Dawson
+44 7900 005 759
ddawson@velti.com

Marcus Kern
+44 7932 661 527
mkern@velti.com