

“Hold this for a moment”

Real-world queuing strategies



David Dawson
Director of Technology

Marcus Kern
VP, Technology



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.



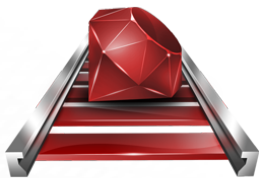
Velti Technologies



- Erlang
- RIAK & leveldb
- Redis
- Ubuntu

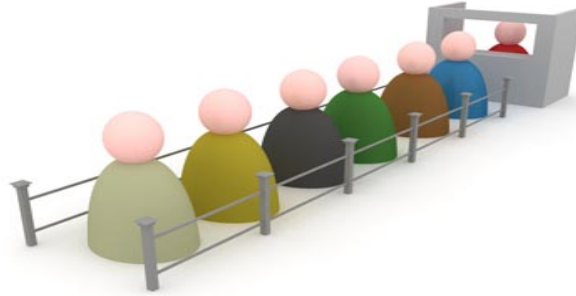


- Ruby on Rails
- Java
- Node.js
- MongoDB
- MySQL

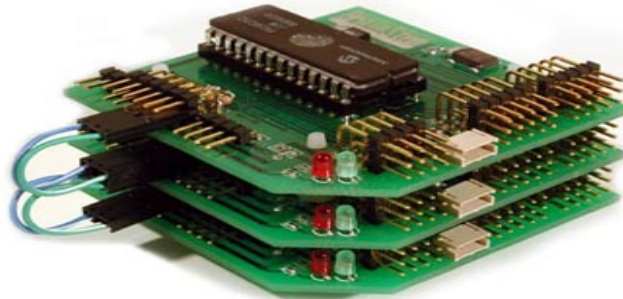


Two parts to this story

- Queuing Strategies



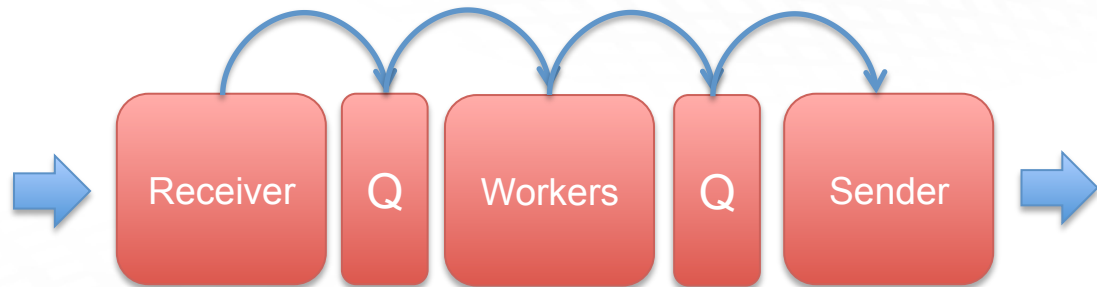
- Optimizing hardware



Background

- Building apps rely on messaging + queuing > 8 years
- Multiple iterations, also combining NoSQL to help scale
- Started with Mysql
 - Operations friendly
 - Although struggled to scale as we our throughputs increased
- What about existing solutions (e.g. RabbitMQ)
 - Features missing until recently
 - More confidence from a iterative approach of our existing design

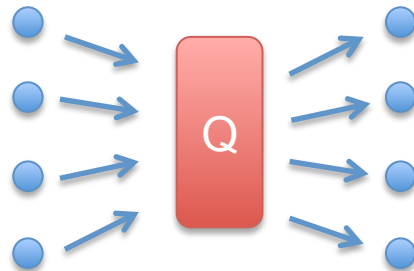
Building a Robust Queue



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

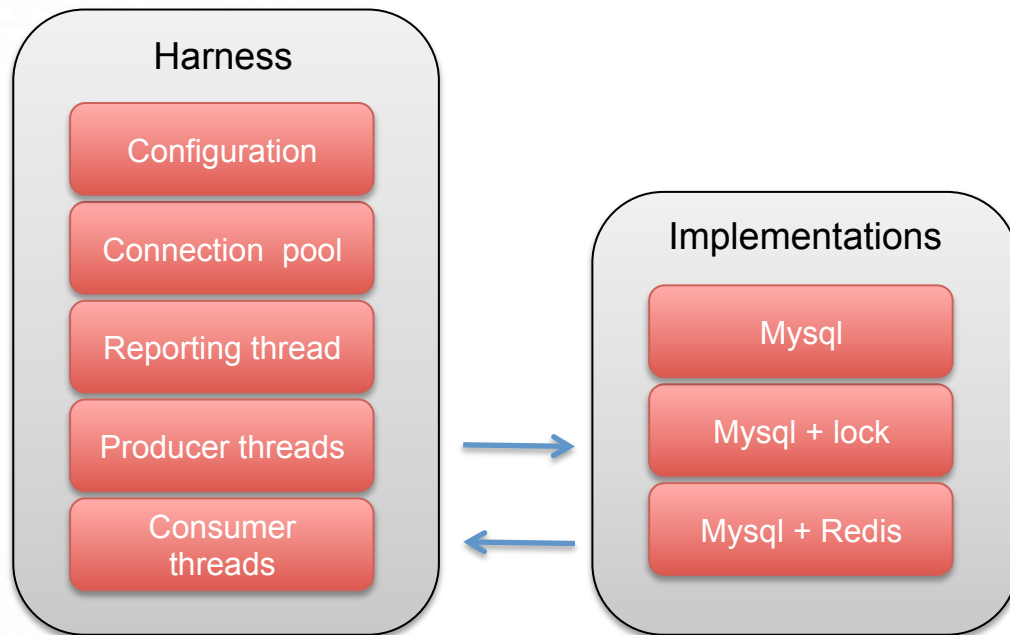
Producers

Consumers



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 iterations
- 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

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

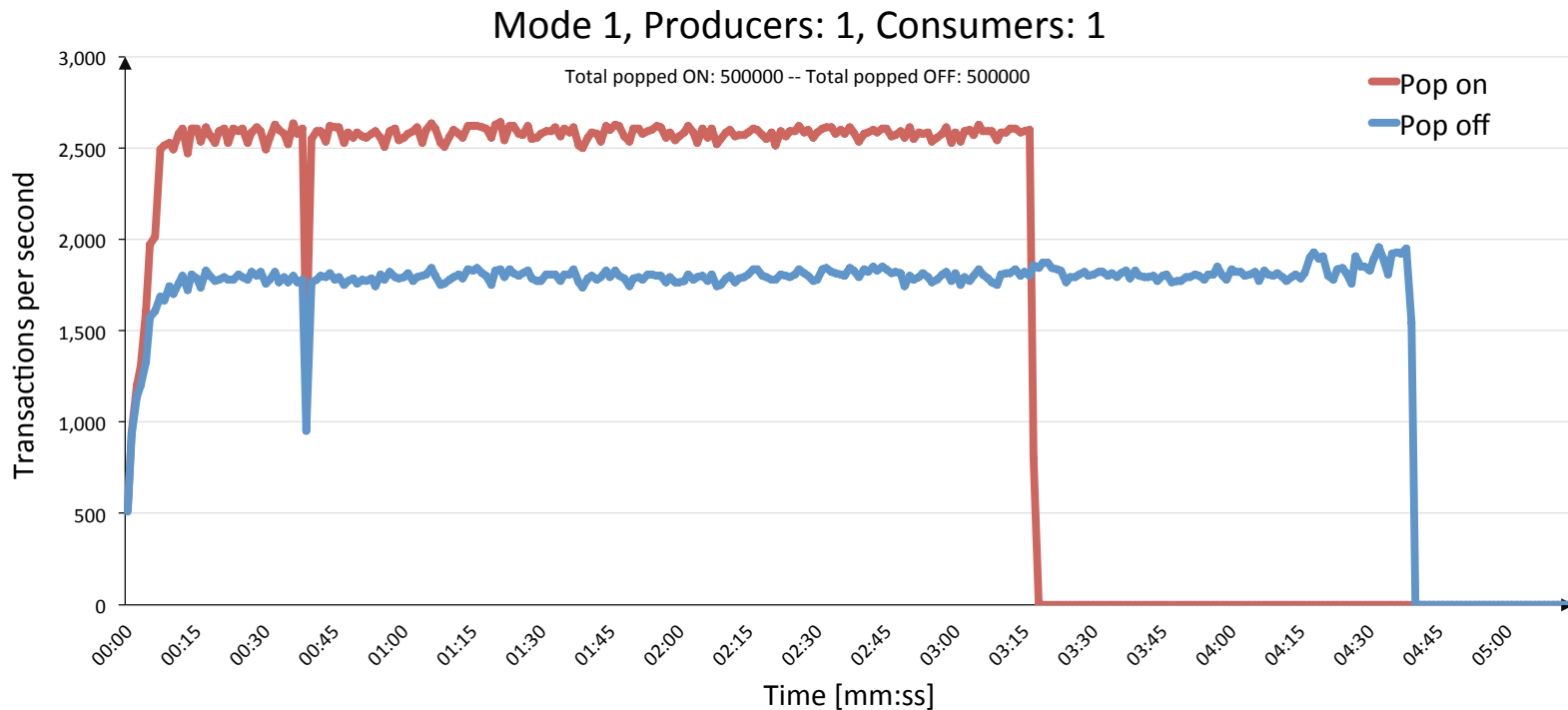
Update queue set worker_id=**123** where
worker_id=0 and process_at <= now() limit 10

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

Update queue set worker_id=-1 where id=2

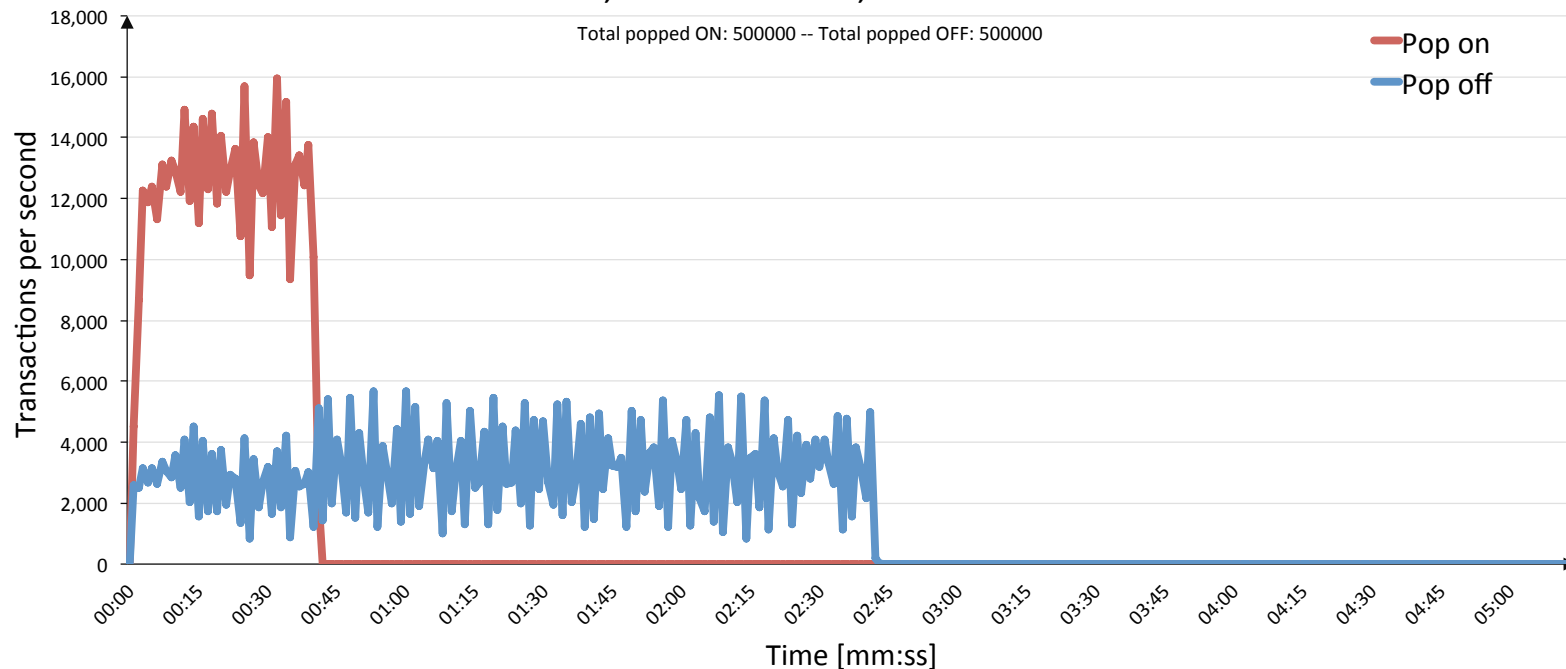
Update queue set worker_id=-1 where id=3

Implementation #1



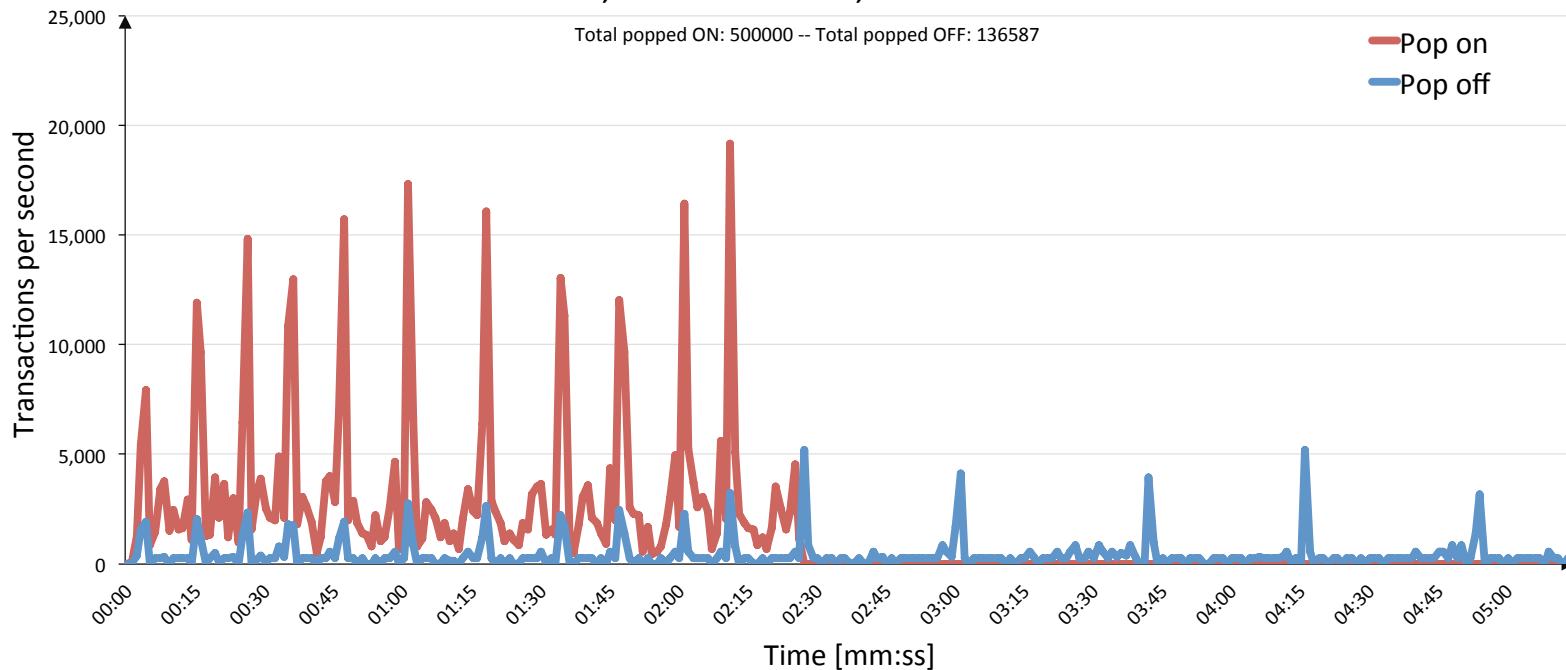
Implementation #1

Mode 1, Producers: 10, Consumers: 10



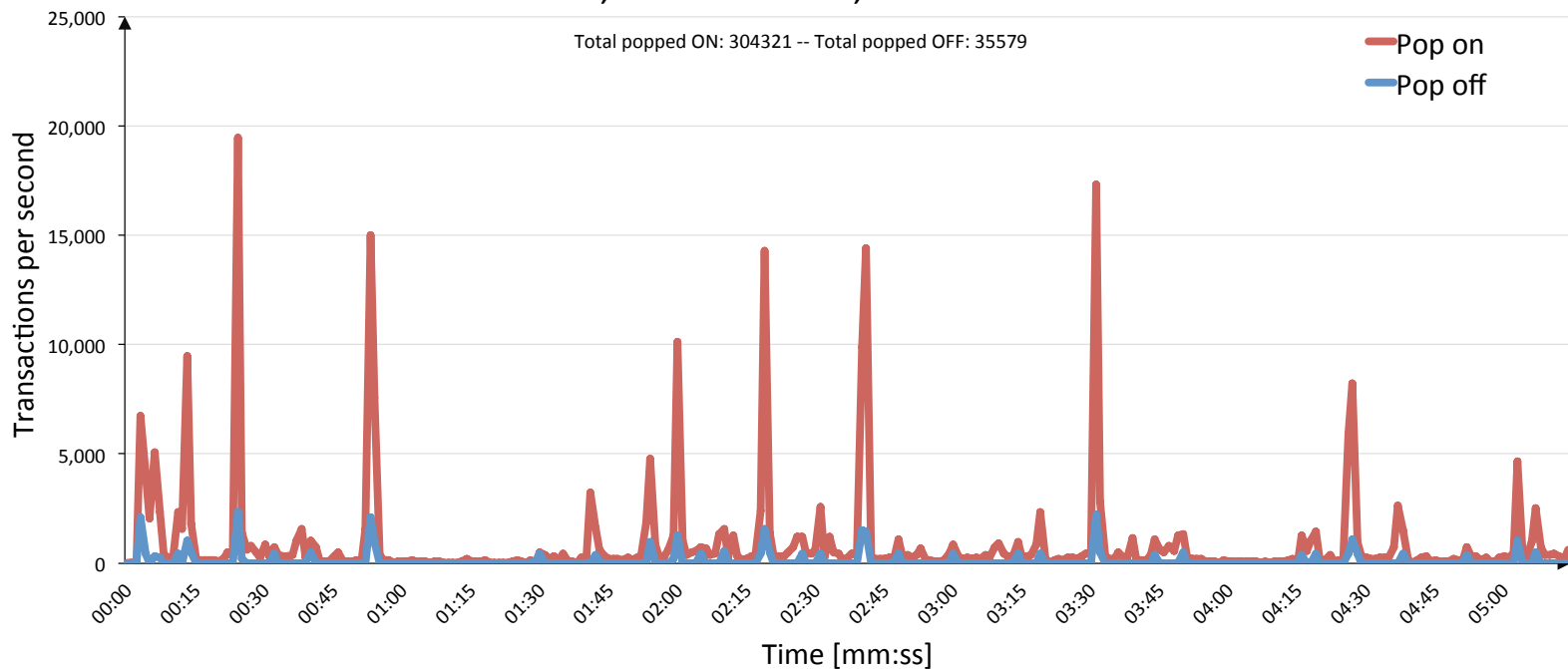
Implementation #1

Mode 1, Producers: 30, Consumers: 30



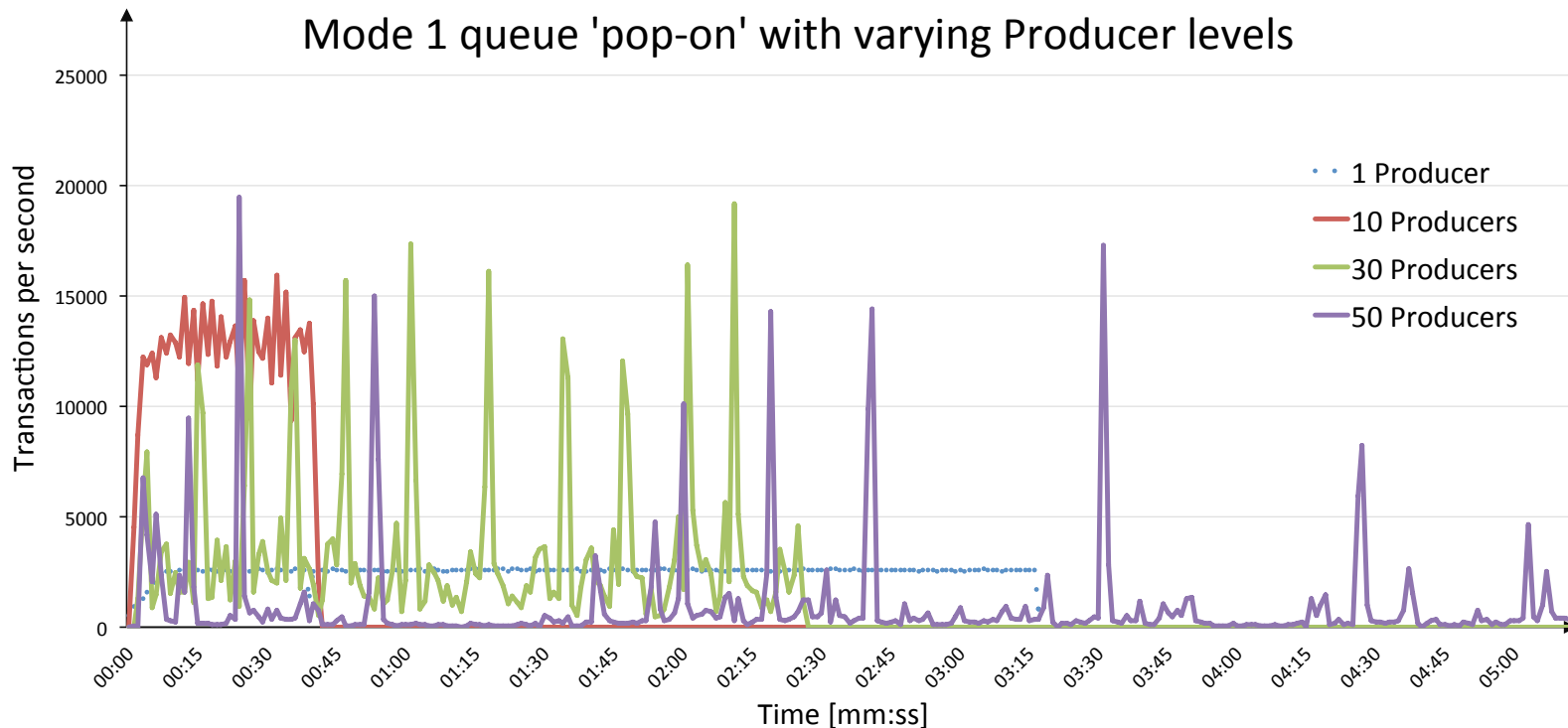
Implementation #1

Mode 1, Producers: 50, Consumers: 50



Implementation #1

Mode 1 queue 'pop-on' with varying Producer levels



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)

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
```

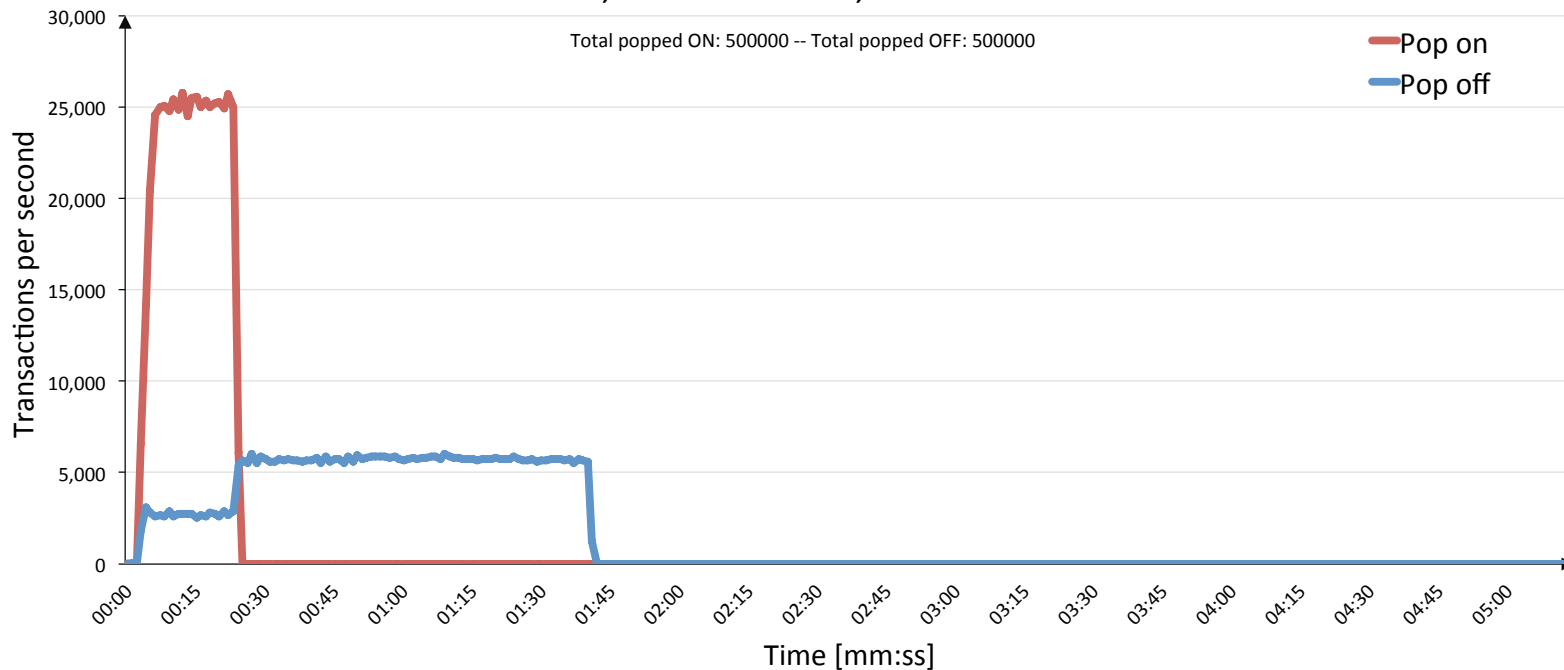
```
Update queue set worker_id=-1 where id=2
```

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

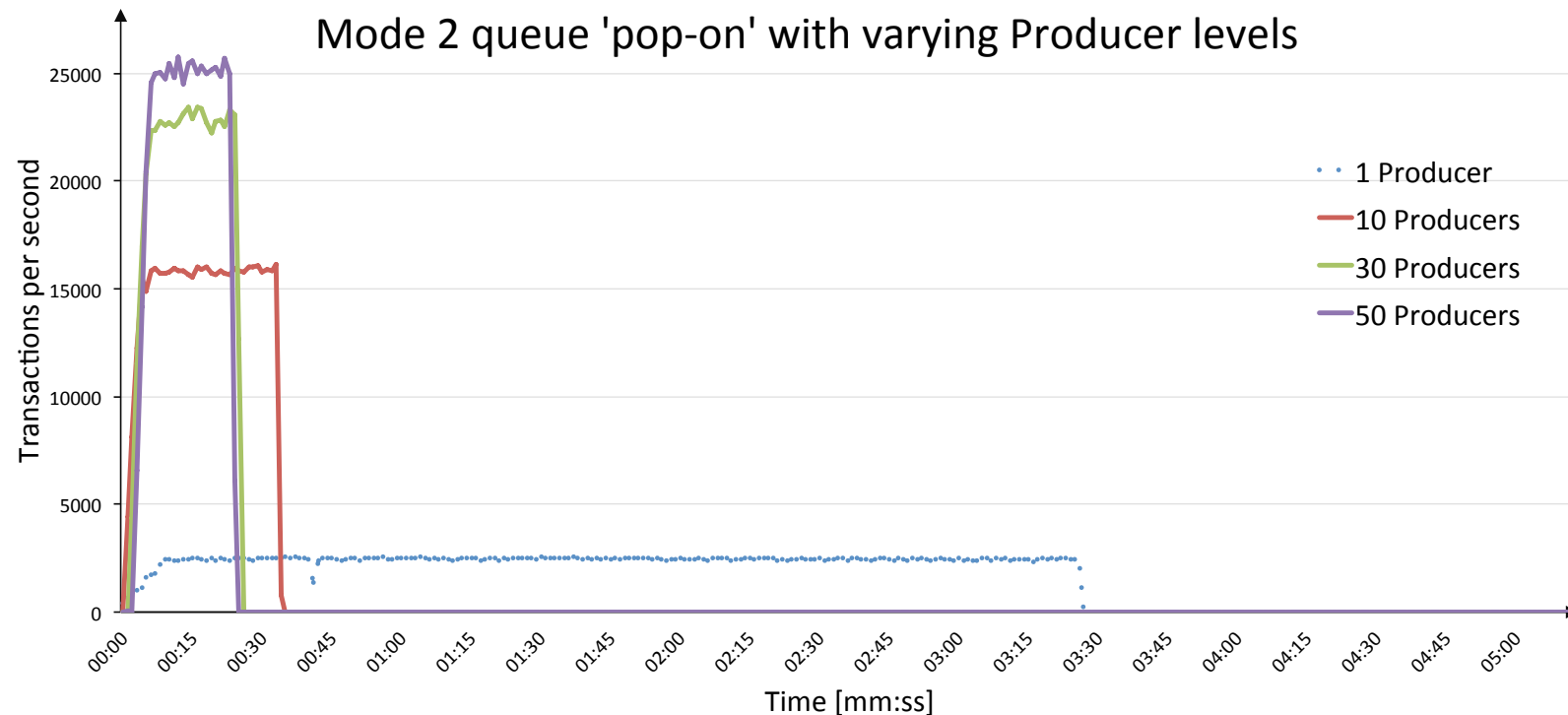


Implementation #2

Mode 2, Producers: 50, Consumers: 50



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 semantics)
- Dedicated hardware
 - Harness: HP DL365 (12 cores)
 - Mysql + Redis: HP DL365 (12 cores)



Implementation #3

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 )
```

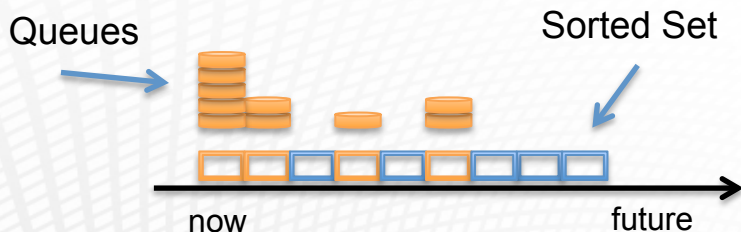
```
Update queue set status='finished' where id = 2
```

```
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

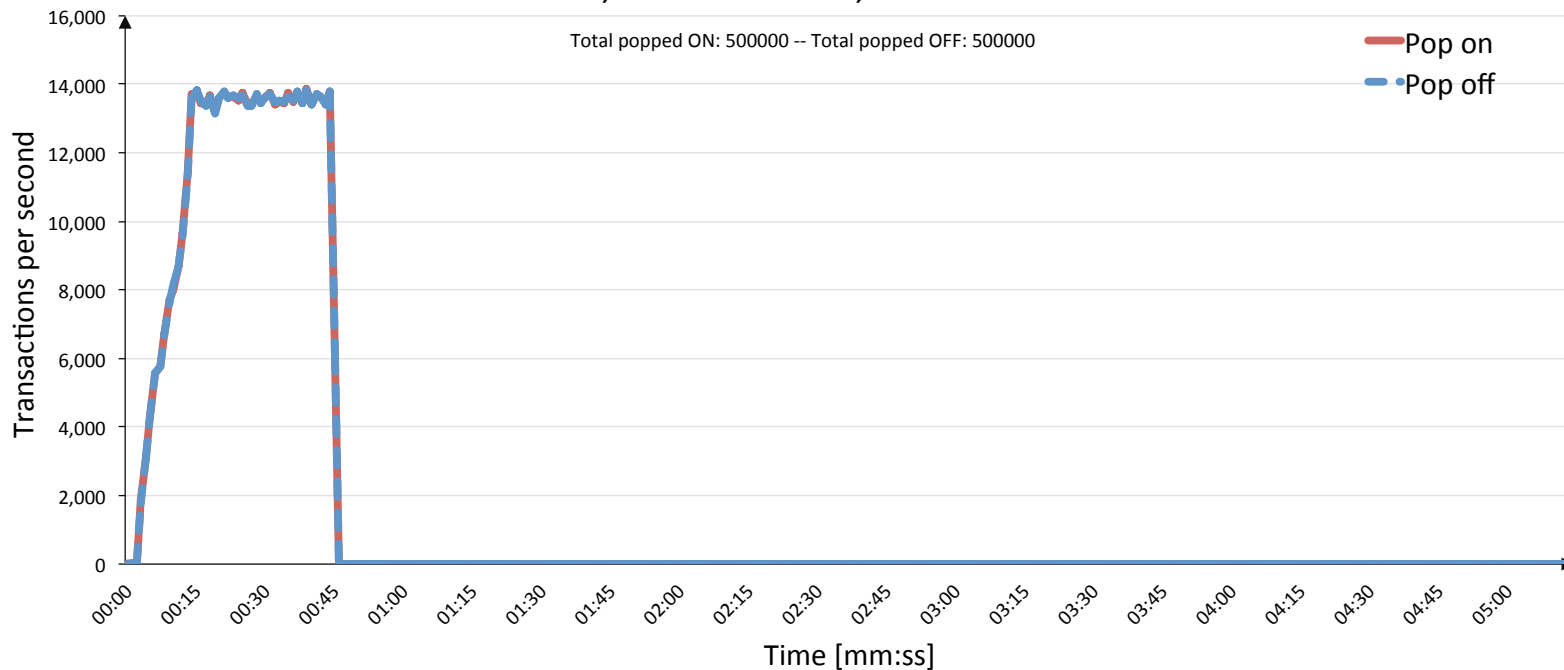
```
queue_name = 'queue' + scheduled_time  
rpush( queue_name, id_of_mysql_insert )  
zadd( 'q_set', scheduled_time, queue_name )
```

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 )
```

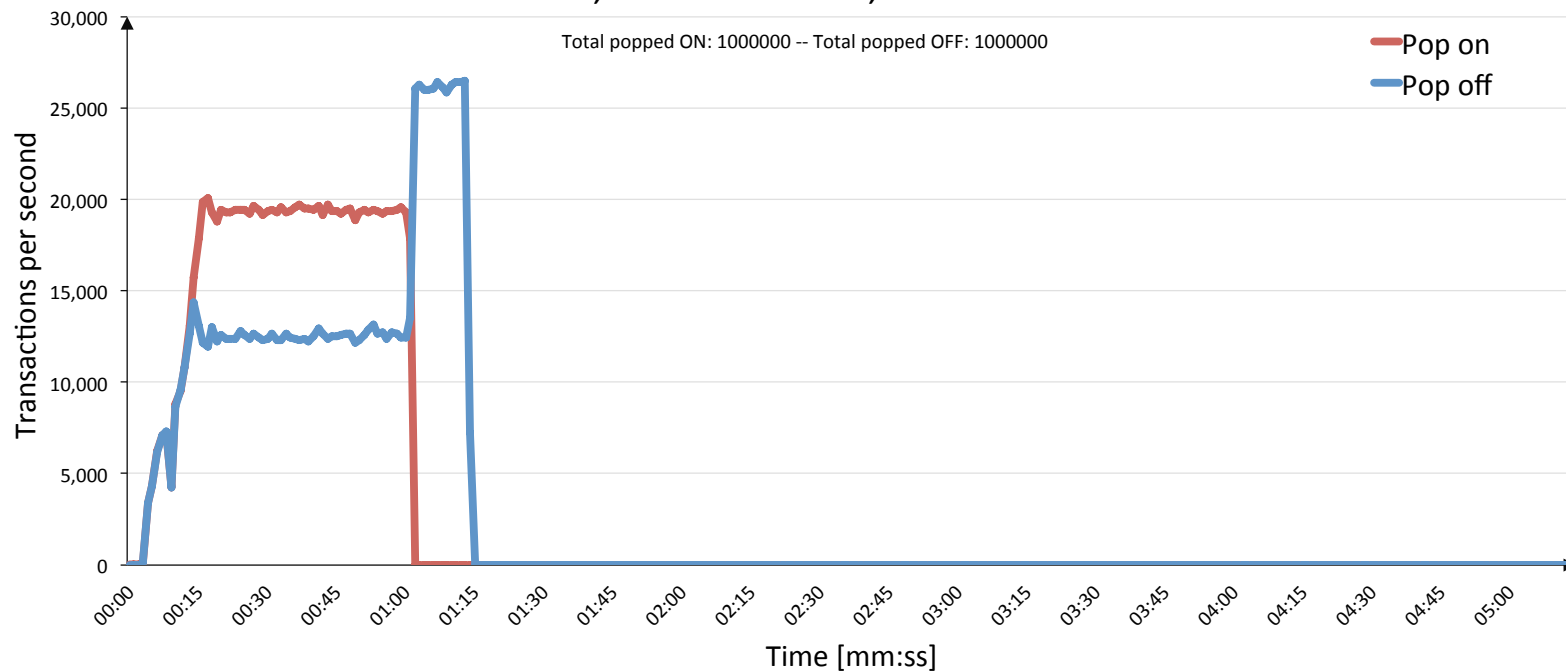
Implementation #3

Mode 3, Producers: 50, Consumers: 50



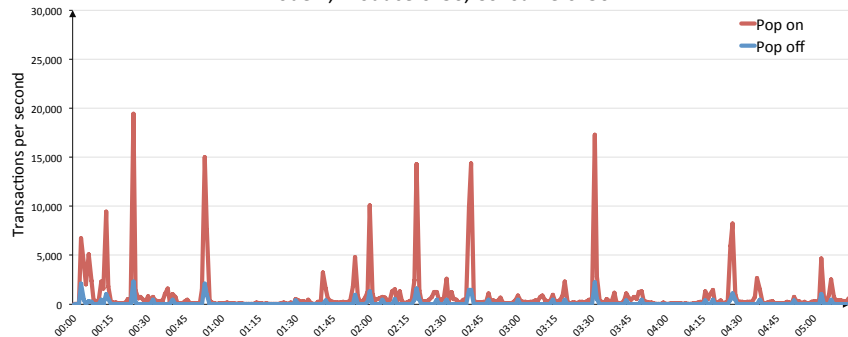
Implementation #3

Mode 3, Producers: 100, Consumers: 50

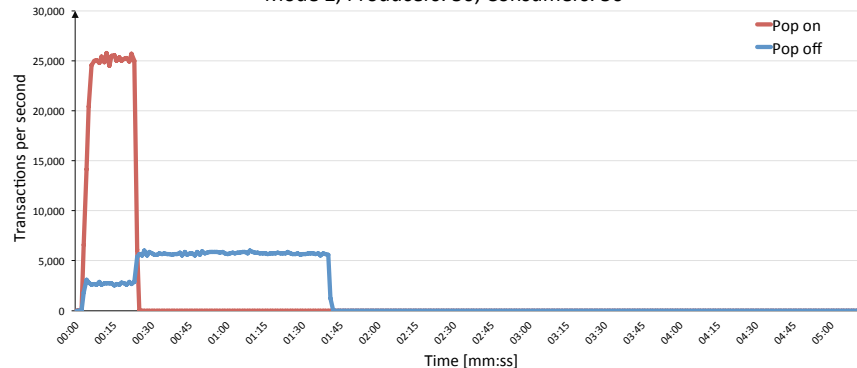


Comparison

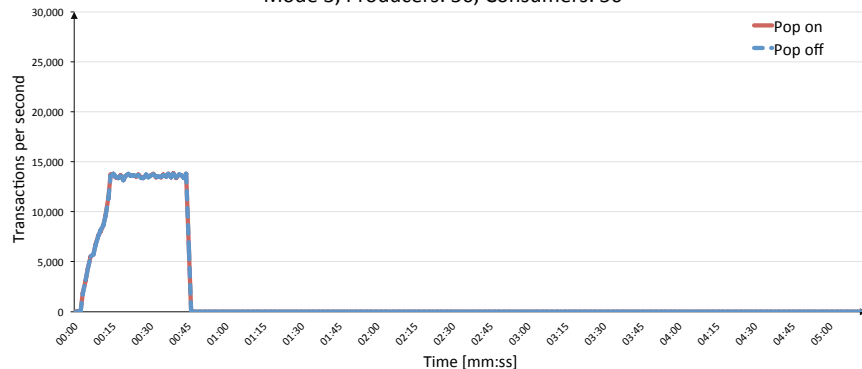
Mode 1, Producers: 50, Consumers: 50



Mode 2, Producers: 50, Consumers: 50



Mode 3, Producers: 50, Consumers: 50



Summarize Results

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

Implementation #3

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

- Most complicated ✗
- Recovery scripts ✗
- Multiple moving parts ✗
- Transactions are hard ✗



+



redis

Future Considerations

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



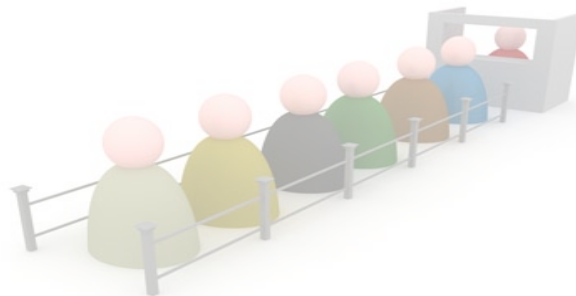
+



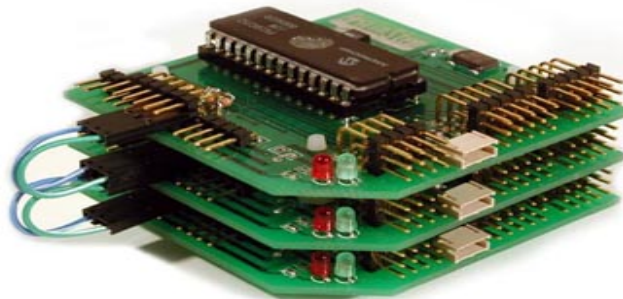
redis

Two parts to this story

- Queuing Strategies



- Optimizing hardware



Hardware optimisation

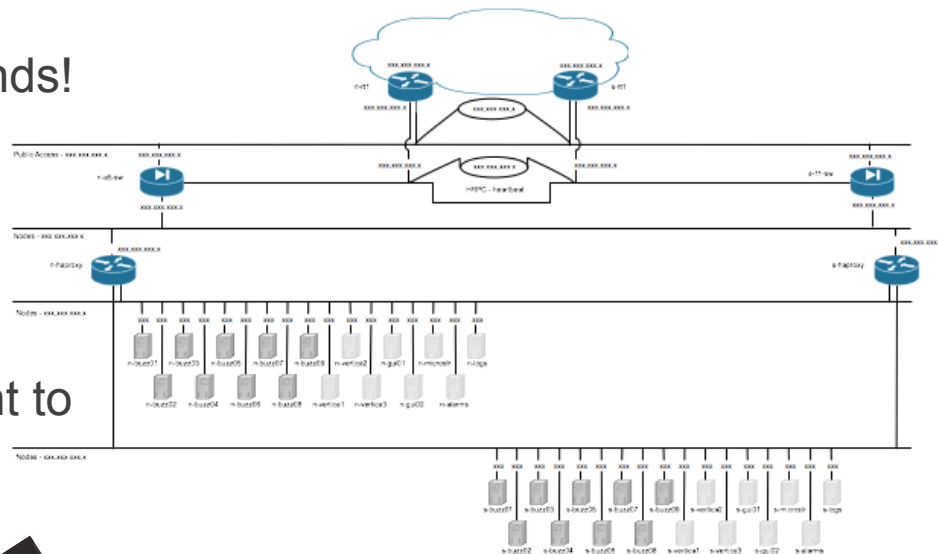
- Observed **'time outs'**
App ⇔ RIAK DB
- Developed sophisticated balancing mechanisms to code around them, but they still occurred
- Especially under load



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

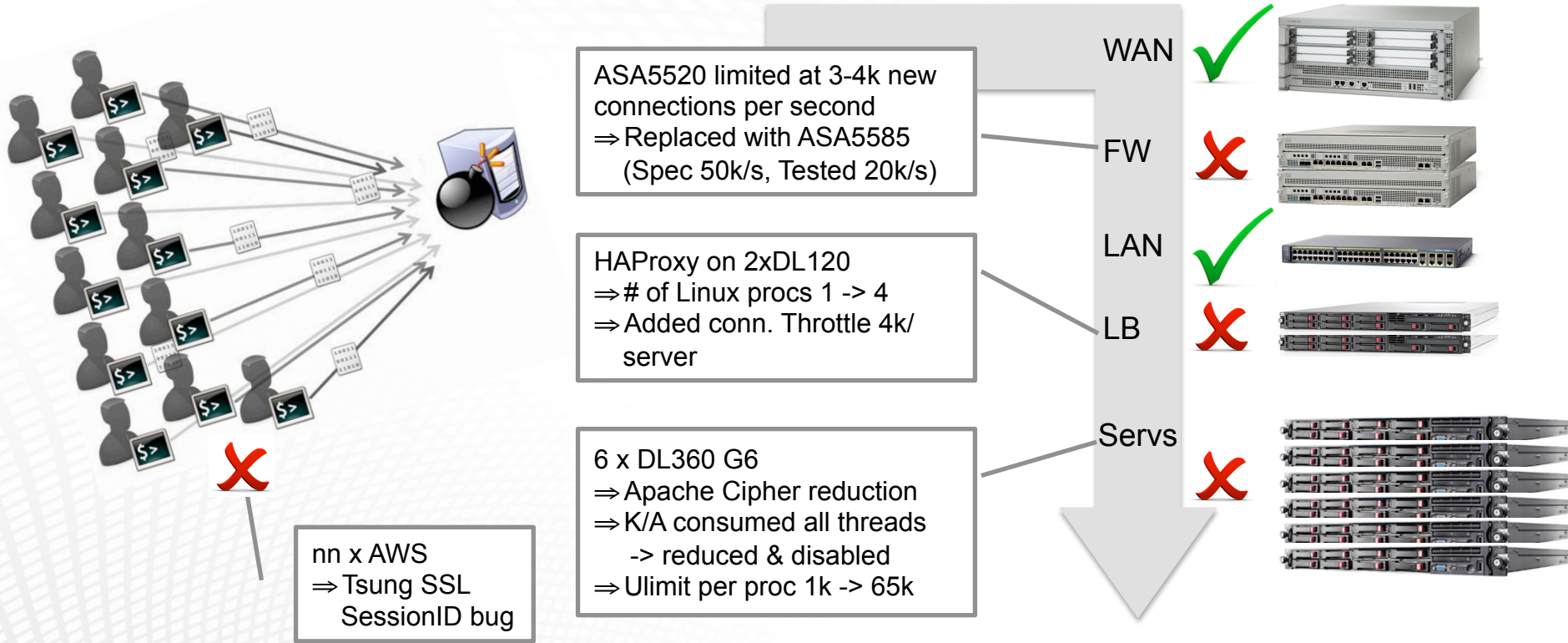
- Processor affinity – Bind a particular process type to a specific processor
- Instruct memory usage to use different memory banks
- For example: `numactl --cpunodebind 1 -interleave all erl`
- Get it here: `apt-get install numactl`

- => 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



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 ✗
- httpperf
 - 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

tsung - Statistics					
Main Statistics					
Name	Highest Value	Lowest Value	Highest Rate	Mean	Count
connect	0.004 msec	0.200 msec	5703.0 / sec	0.320 msec	164704
page	98.32 msec	2.10 msec	5703.0 / sec	2.22 msec	164704
request	0.432 msec	0.283 msec	36802.8 / sec	0.307 msec	62101054
session	1200.4 sec	1200.0 sec	416.0 / sec	1200.0 sec	443009
Transactions Statistics					
Name	Highest Value	Lowest Value	Highest Rate	Mean	Count
ts_http	0.101 sec	0.000 msec	890.8 / sec	0.000 msec	108484
Network Throughput					
Name	Highest Rate	Total			
ts_http	20.10 MB/sec	10.44 GB			
ts_https	20.10 MB/sec	10.44 GB			
Counters Statistics					
Name	Highest Rate	Total number			
ts_http	415.0 / sec	48000			
ts_https	3.1 / sec	312			
ts_https	481 / sec	110576			
Counters Statistics					
Name	Max				
connected	10100				
sources	100100				



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:

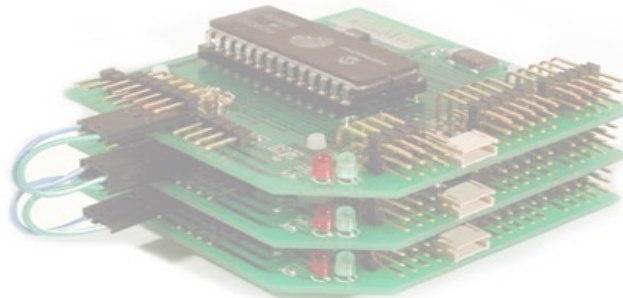
[git@github.com:mitadmin/ionstorm.git](https://github.com/mitadmin/ionstorm.git)

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 :

David Dawson

+44 7900 005 759

ddawson@velti.com

Marcus Kern

+44 7932 661 527

mkern@velti.com

Building a wallet

- Fast

- Over 1,000 credits / sec
- Over 10,000 debits / sec (votes)



- Scalable

- Double hardware == Double performance



- Robust / Recoverable

- Transactions can not be lost
- Wallet balances recoverable in the event of multi-server failure



- Auditable

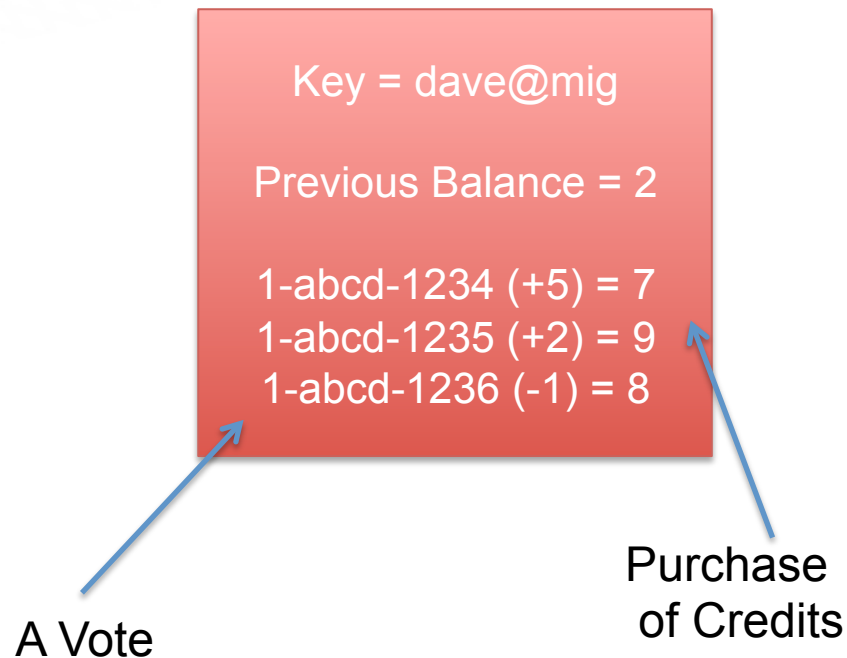
- Complete transaction history



Building a wallet - attempt #1



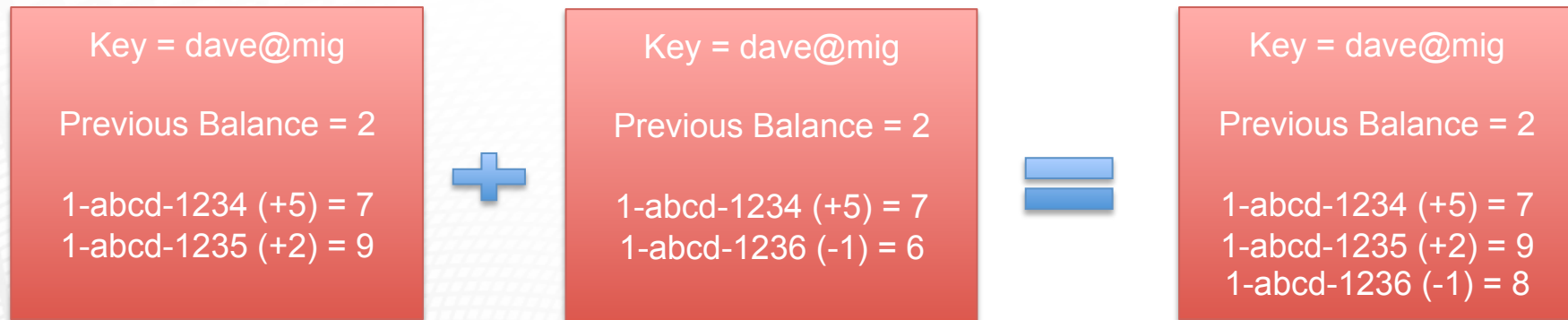
- Use RIAK Only
 - Keep things simple
 - Less moving parts
- A wallet per user containing:
 - Previous Balance
 - Transactions with unique IDs
 - Rolling Balance
 - Credits (facebook / itunes)
 - Debits (votes)



Building a wallet - attempt #1



- RIAK = Eventual Consistency
 - In the event of siblings
 - Deterministic due to unique transactions ID's
 - Merge the documents and store



Building a wallet - attempt #1



- Compacting the wallet
 - Periodically
 - In event it grows to large

Key = dave@mig

Previous Balance = 2

1-abcd-1234 (+5) = 7

1-abcd-1235 (+2) = 9

1-abcd-1236 (-1) = 8

...

1-abcd-9999 (+1) = 78



Compactor



Key = dave@mig

Previous Balance = 78

Building a wallet - attempt #1

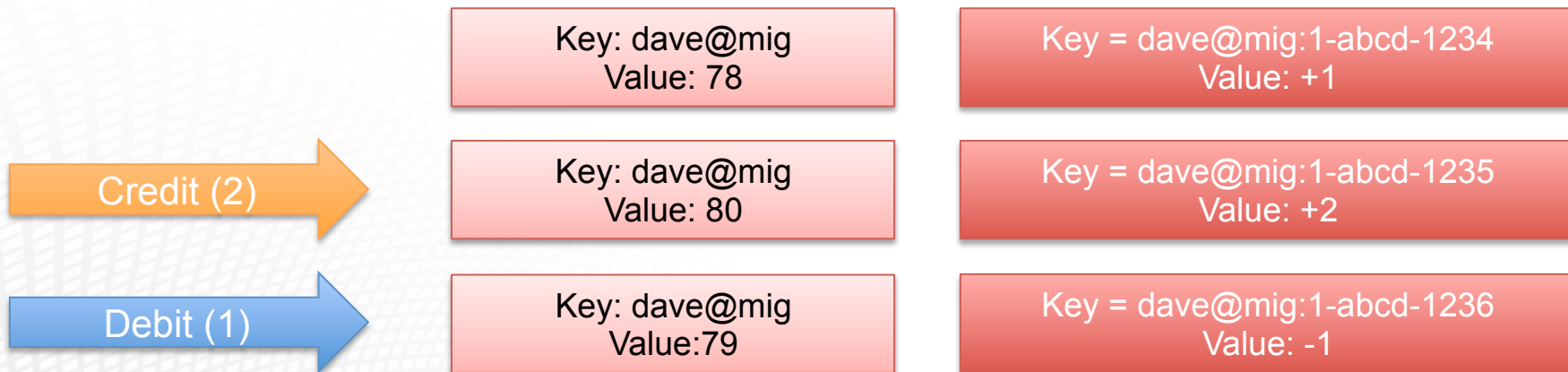


- Our experiences
 - Open to abuse
 - As wallet grows, performance decreases
 - Risk of sibling explosion
 - User can go over drawn

Building a wallet - attempt #2



- Introduce REDIS
 - REDIS stores the balance
 - RIAK stores individual transactions



Building a wallet - attempt #2



- Keeping it all in sync
 - Periodically compare REDIS and RIAK
- Disaster Recovery
 - Rebuild all balances in REDIS
 - Using transactions from RIAK

Building a wallet - attempt #2



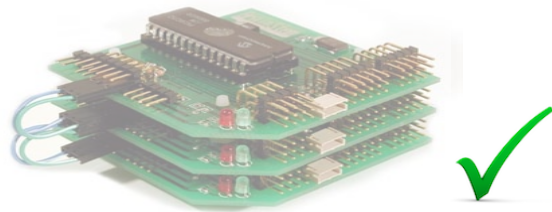
- Our experiences
 - It works
 - Fast 10,000 votes / sec (6 x HP DL385)
 - Used wallet recovery (Data Center Power Fail)
- The future
 - Possible use of levelDB backend for RIAK
 - Faster wallet recovery

Battle Stories #2

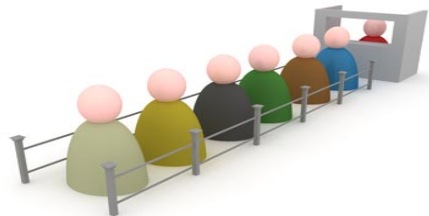
- Building a wallet



- Optimizing your hardware stack



- Building a robust queue – final version



Building a Queue

- Fast
 - > 1000 msg /sec
- Scalable
 - Double the machines, double the capacity
- Recoverable
 - In the event of a failure, all messages can be recovered

Design

- Queues stored in memory (volatile)
 - Hand rolled our own using ETS (erlang)
 - We needed to add complex behavior such as scheduling
 - Overflow protection by paging to disk
- Copy of the data and state stored in a shared data store
 - RIAK ticked all the boxes
 - Scalable
 - Robust
 - Fast

Previously

- We explored RIAK to store and recover the queues using:
 - Index's (levelDB)
 - Latencies too unpredictable
 - Performance was less than half of bitcask
 - Key Filtering (bitcask)
 - Write overhead too expensive as we had to update the key not the value (delete and insert)
 - Real world performance under load was not great
 - Map Reduce across all key (bitcask)
 - Great for small data sets
 - Forget it as your data set get's into the 10 of millions

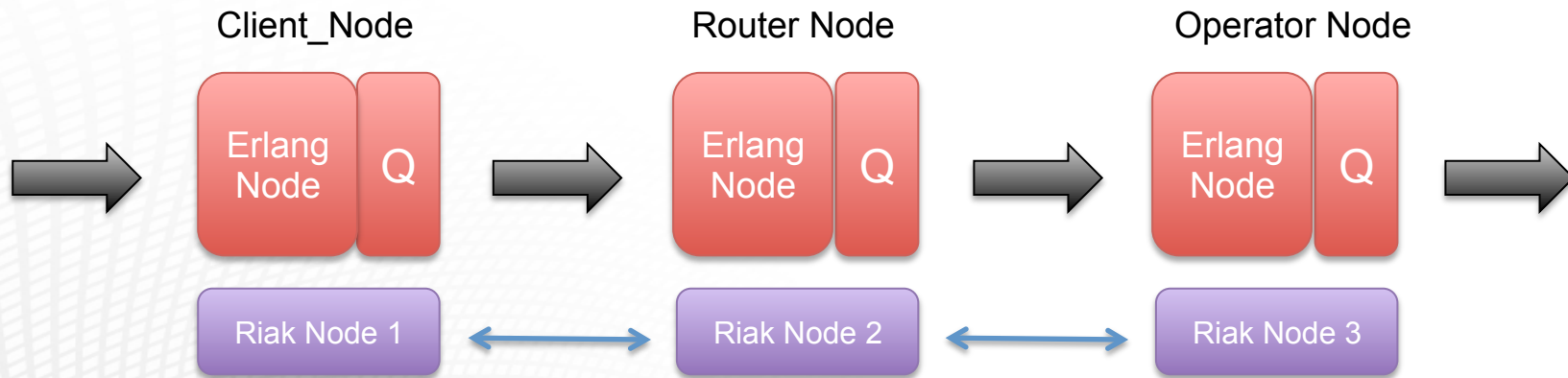
New Approach

- With a little help from the Basho guys we came up with a new approach
- Predictable keys + Snapshots (bitcask)
 - Simple
 - Smallish impact on performance
 - It worked
 - And it scales

Our Architecture

- Each Node has it's own Queue
- Each Node lives on it's own physical machine
- RIAK runs as a cluster on all of the nodes

Basic SMS Gateway topology



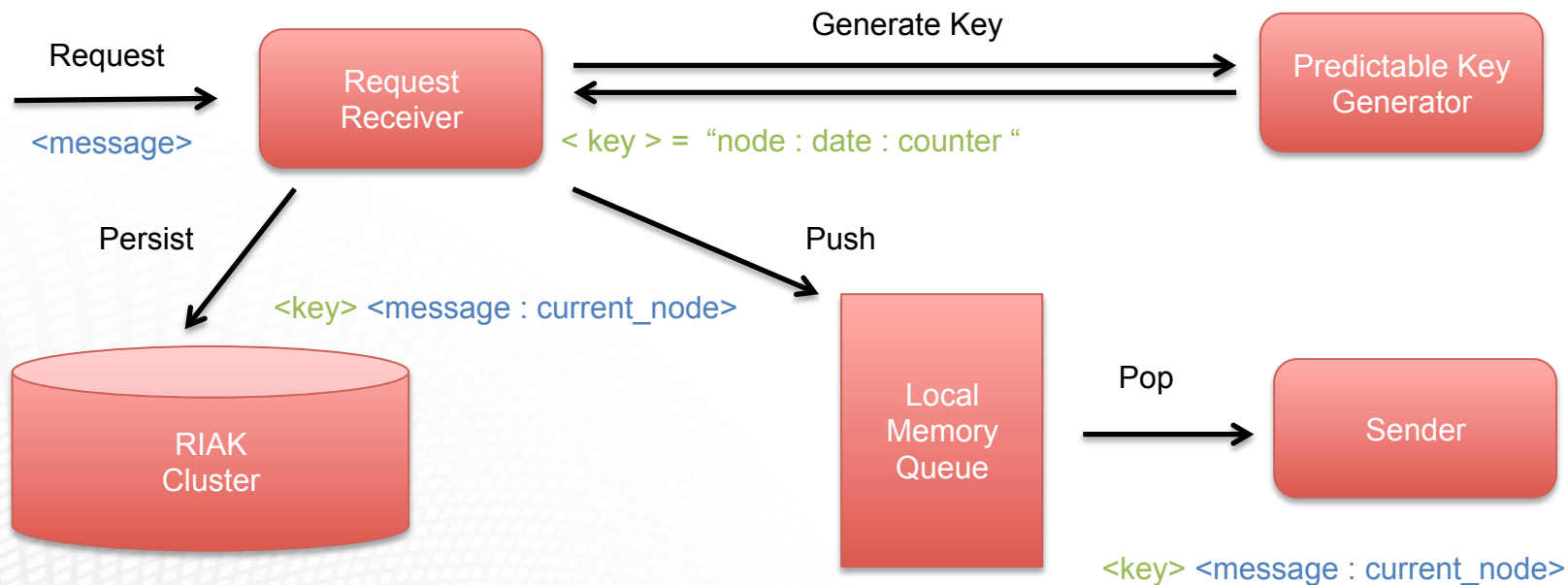
Predictable Key

- Key: “node : date : counter “
 - **node:** the name of the originating node for the request e.g “client_node”
 - **date:** e.g. “2012-01-01”
 - **counter:** number of message since the beginning of the day. “3000”
- Value: <message : current_node >
 - **message:** the original request e.g. “send sms”
 - **current_node:** the current node the message is located e.g. “router_node”

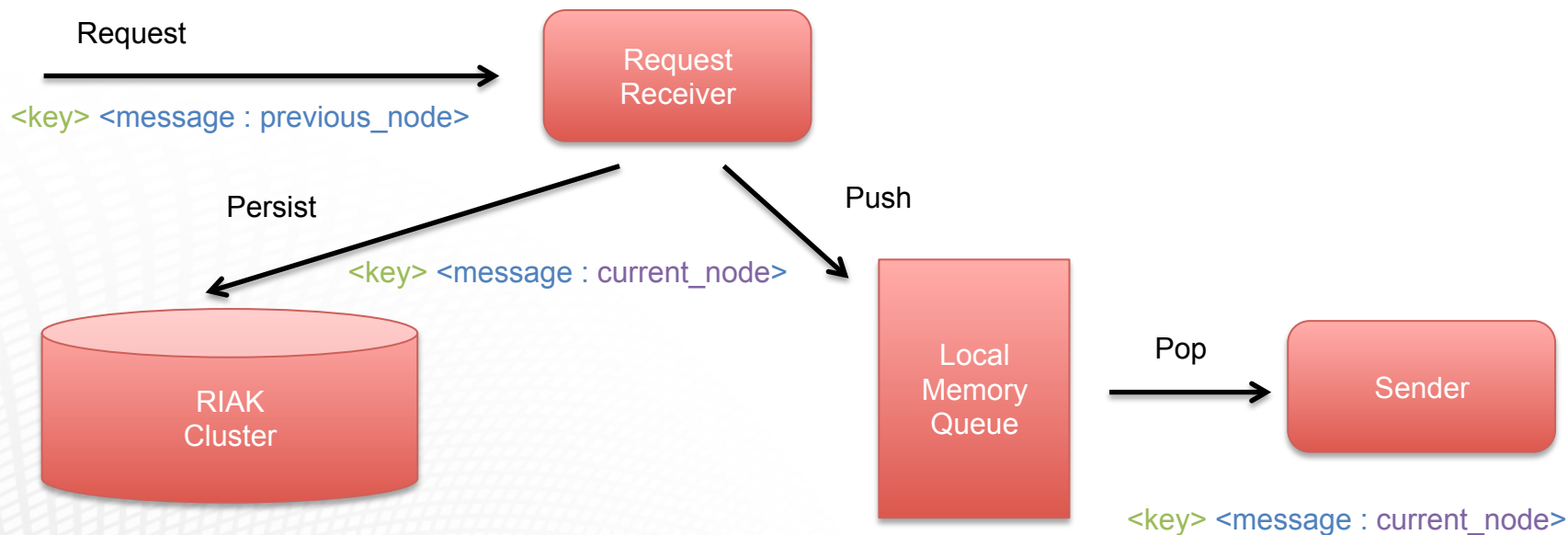
Snapshot

- Every 1000 messages
- Take a snapshot of the counter
 - Key: “client_node : 2012-01-01 : snapshot ”
 - Value: 5000
- This is then used to help determine an upper limit for the recovery
 - Which will be discussed in more detail in a couple of slides

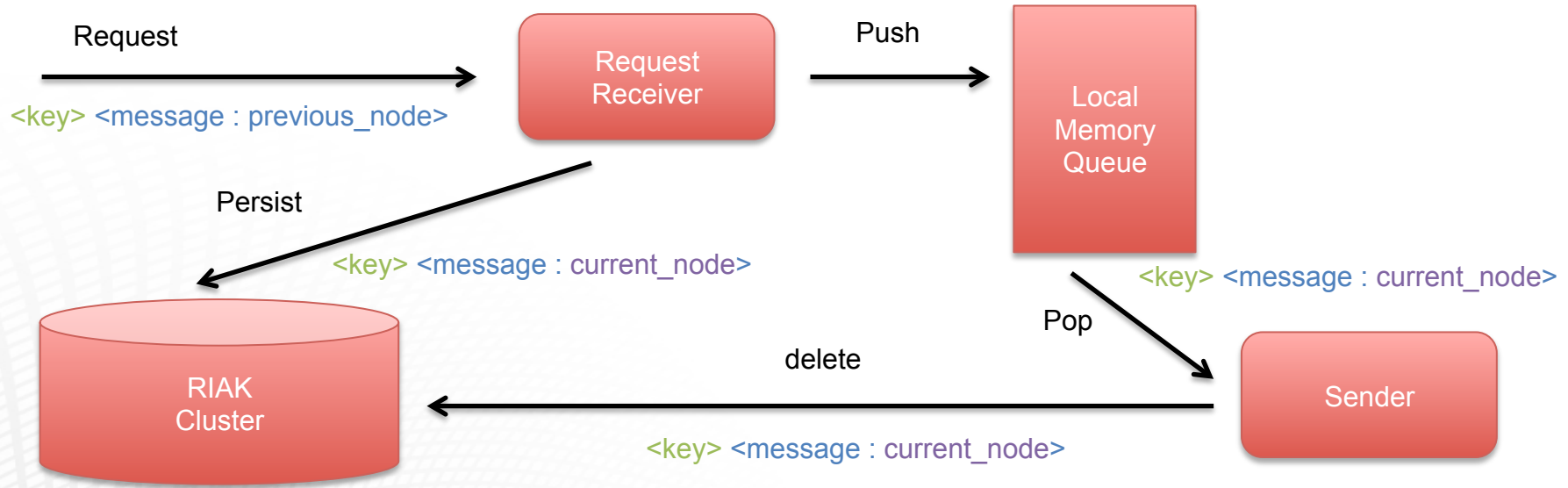
Queue – incoming node



Queue – intermediate node



Queue – outgoing node



Recovery

- Take all originating nodes e.g. “client_node”
- Using the current date e.g. “2012-01-01”
- Use the snapshot to get the last current count recorded e.g. “3000”
 - Key: “client_node : 2012-01-01 : snapshot ”
 - Value: 3000
- Rebuild by walking the keys from:
 - from the value: 1
 - to the current count + (2 x snapshot interval): 5000
 - Across all originating nodes and dates < 5 days

Testing

- Benchmarking with 3 x HP365's (AMD)
 - Production has 18 x HP360's
- Sustained 2000 req/sec (8 x RIAK ops per request)
 - Linear scaling in testing
- Recovered 5 million messages in < 1 hour after crashing a node
 - Whilst processing 500 req/sec sustained

Production



- Currently live and used for our SMS Gateway
- No noticeable drop in performance when under peak loads
- Plan to be used in our other products
- Hopefully our final solution