Working with Velti

- Our robust technology has been used by major broadcasters and media clients for over 7 years
- Voting, Polling and Real-time Interactivity through second screen solutions
- Incremental revenue generating services integrated with TV productions
- Facilitate 10,000+ interactions per second as standard across our platforms
- Platform and services have been audited by Deloitte and other compliant bodies
- High capacity throughput for interactions, voting and transactions on a global scale
- Partner of choice for BBC, ITV, Channel 5, SKY, MTV, Endemol, Fremantle and more:

BBC
itv
CHANNEL5
sky
5 TELECINCO
7
MTV MUSIC TELEVISION
O2
THE X FACTOR
FREMANTLEMEDIA
endemol
SONY
f
vodafone

mVoy/mGage Products

mVoy™ connect — High volume mobile messaging campaigns & mobile payments

mVoy™ engage — Social Interactivity & Voting via Facebook, iPhone, Android & Web

mVoy™ publish — Create, build, host & manage mobile commerce, mobile sites & apps

mVoy™ communicate — Interactive messaging & multi-step marketing campaigns

Velti Technologies

- Erlang
- RIAK & leveldb
- Redis
- Ubuntu

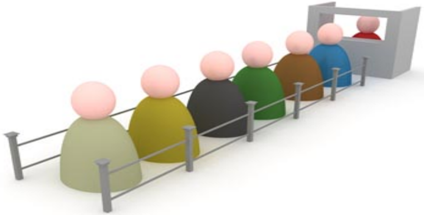
- Ruby on Rails
- Java
- Node.js
- MongoDB
- MySQL

Battle Stories #2

(- Building a wallet)

- Optimizing your hardware stack

- Building a robust queue



Building a wallet

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

- Scalable
  - Double hardware == Double performance

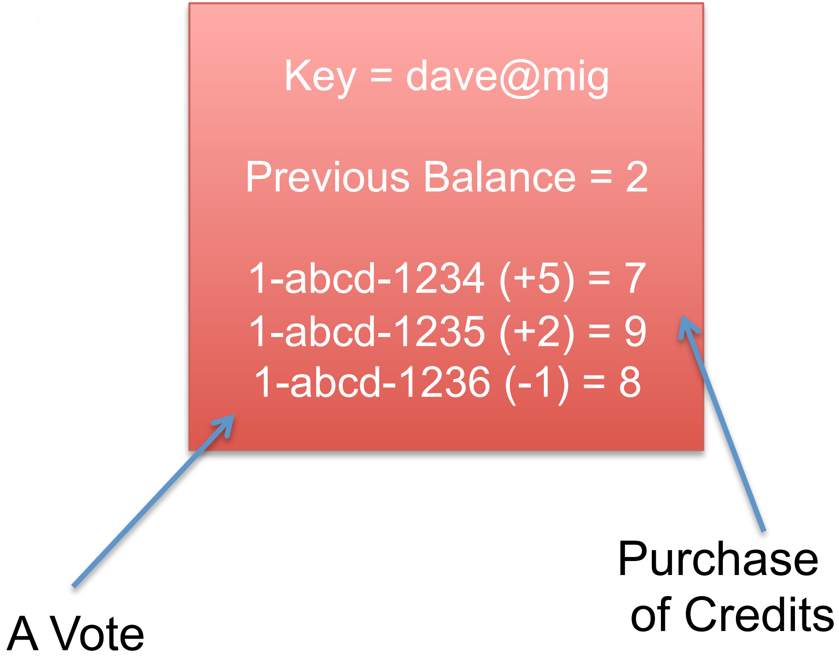
- Robust / Recoverable
  - Transactions can not be lost
  - Wallet balances recoverable in the event of multi-server failure

- 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)



Key = dave@mig

Previous Balance = 2

1-abcd-1234 (+5) = 7

1-abcd-1235 (+2) = 9

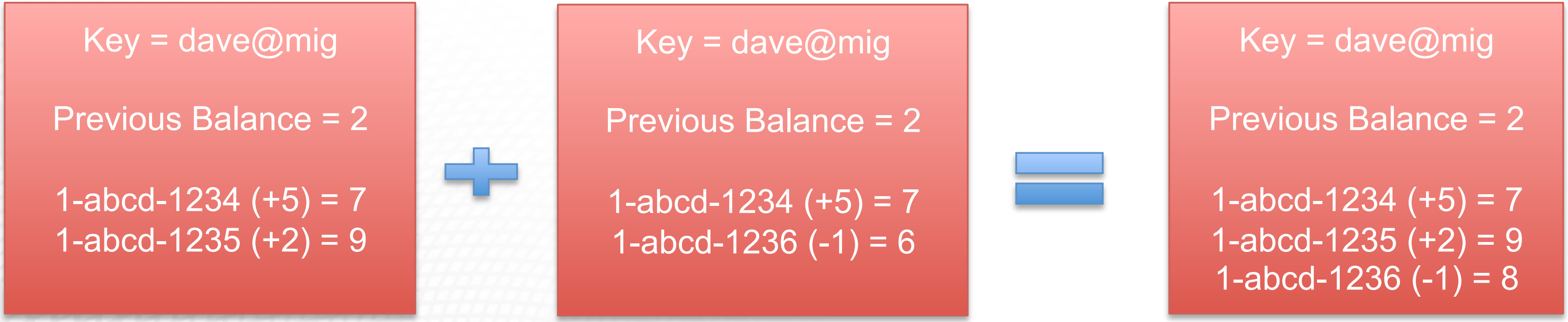
1-abcd-1236 (-1) = 8

A Vote

Purchase of Credits

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



Key = dave@mig

Previous Balance = 2

1-abcd-1234 (+5) = 7
1-abcd-1235 (+2) = 9

Key = dave@mig

Previous Balance = 2

1-abcd-1234 (+5) = 7
1-abcd-1236 (-1) = 6

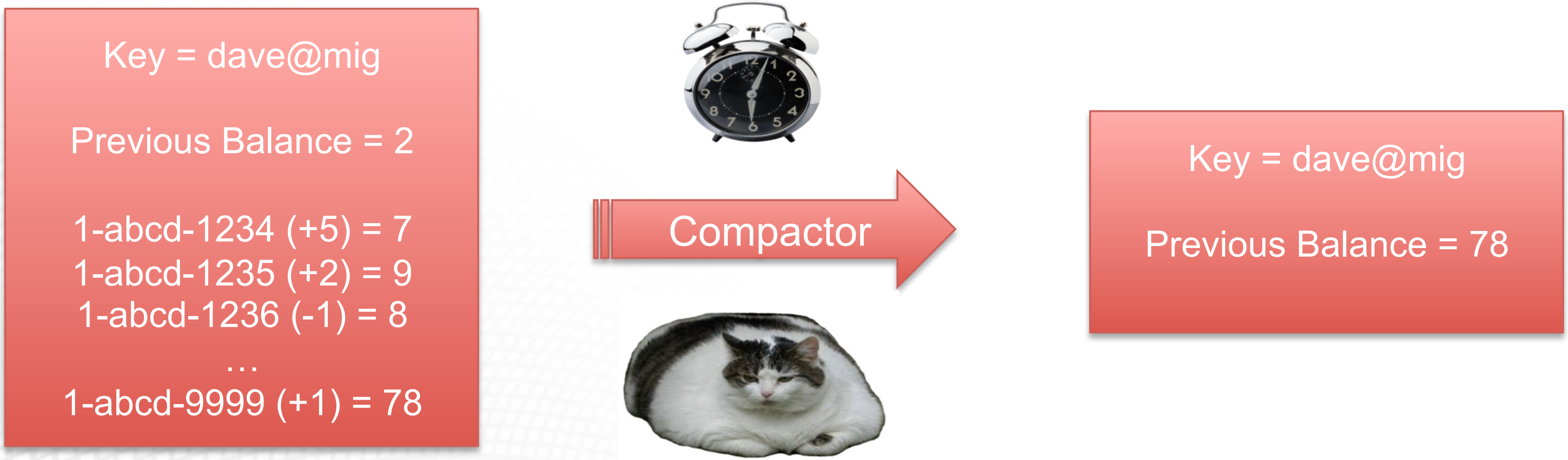
Key = dave@mig

Previous Balance = 2

1-abcd-1234 (+5) = 7
1-abcd-1235 (+2) = 9
1-abcd-1236 (-1) = 8

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

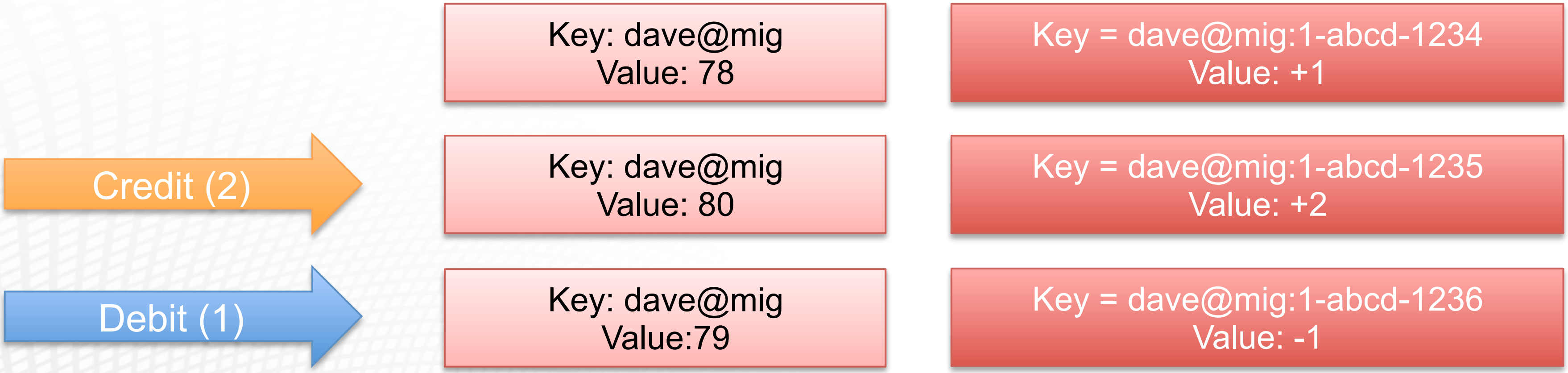
Previous Balance = 78

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



Key: dave@mig
Value: 78

Key = dave@mig:1-abcd-1234
Value: +1

Credit (2)

Key: dave@mig
Value: 80

Key = dave@mig:1-abcd-1235
Value: +2

Debit (1)

Key: dave@mig
Value:79

Key = dave@mig:1-abcd-1236
Value: -1

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

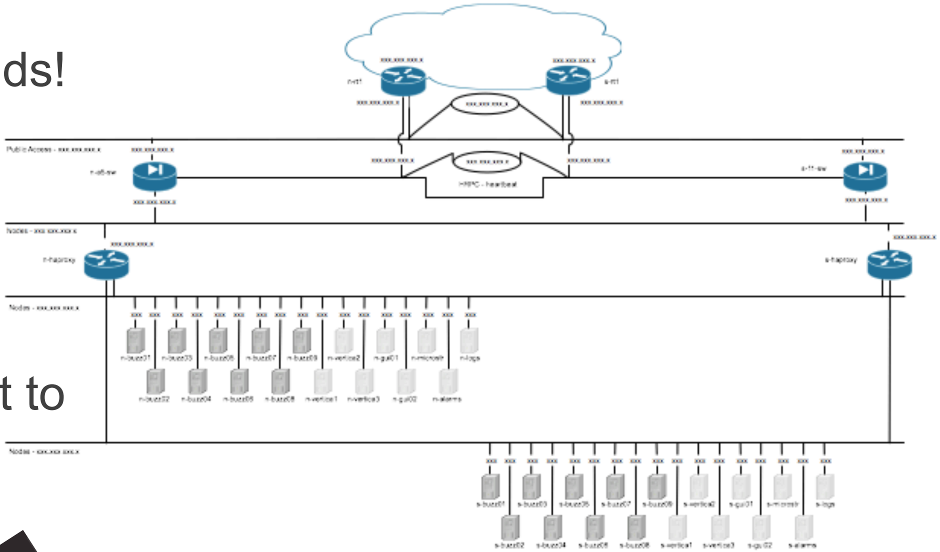
Hardware optimisation

- 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)



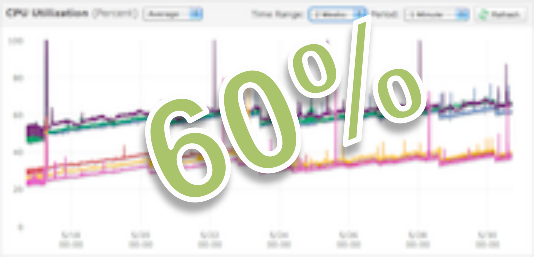
Looking for contention…

Contention options

- CPU

✓

Less than
60%
utilisation



- Disk IO

✓

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

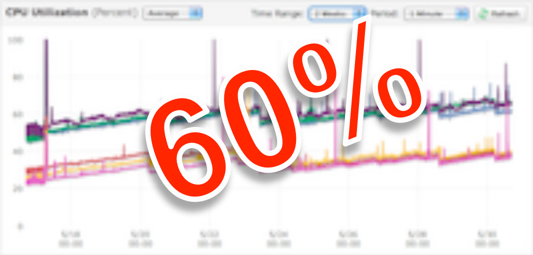
- Network IO

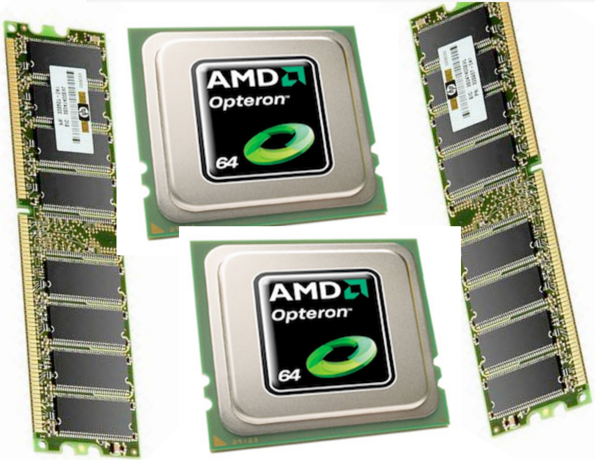
✓

- 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. => ccNUMA
- Linux threading allocation is challenged
- Cache-coherence attracts significant overheads, especially for processes in quick succession!





Gain control! - NUMACTL

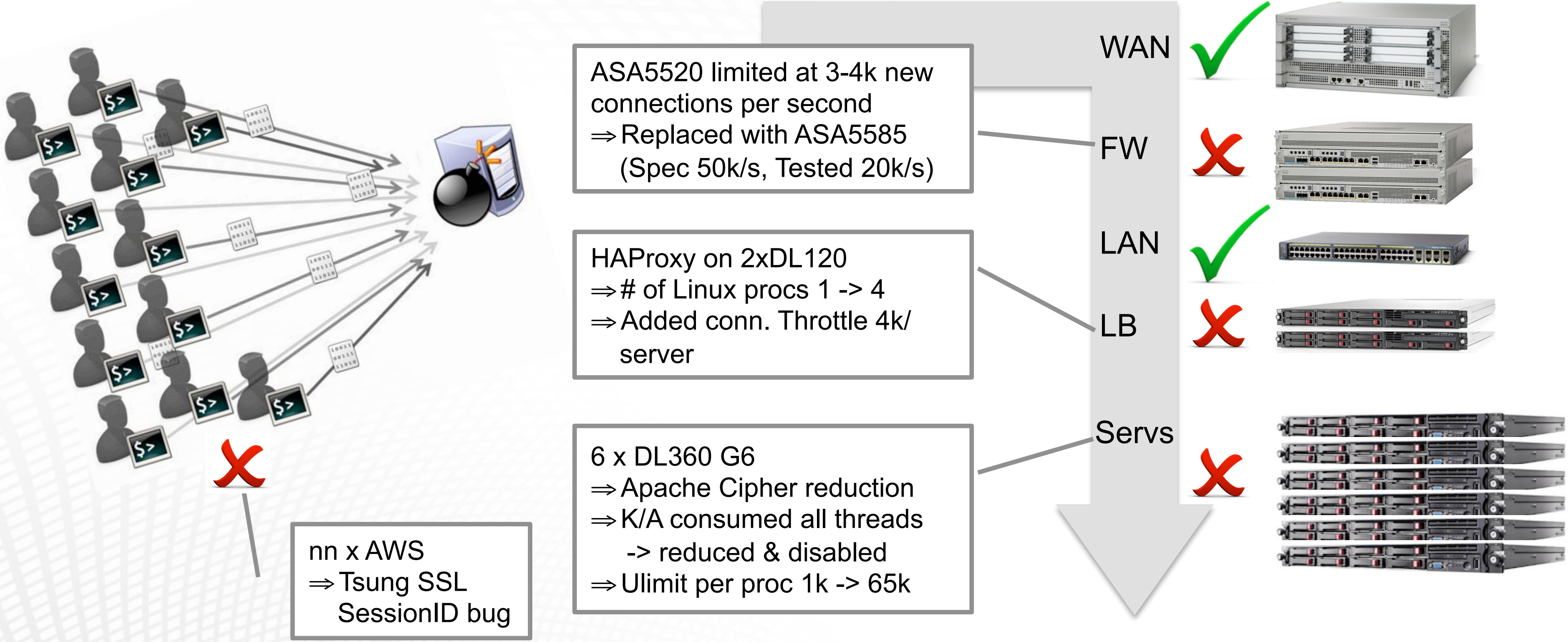
- Processor affinity – Binds a particular process type to a specific processor
- Instruct memory usage to use different banks
- For example: numactl --cpunodebind 1 –interleave all erl
- Get it here: apt-get install numactl

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

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



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

nn x AWS
⇒ Tsung SSL SessionID bug

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

velti

- 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

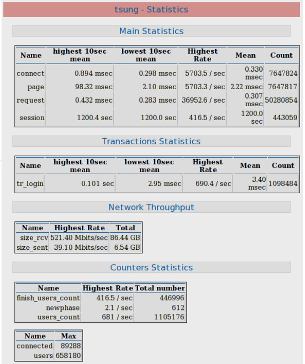
- What is it?
  - Tsung is an open-source multi-protocol distributed load testing tool
  - Written in erlang
  - Can support multiple protocols
    - HTTP / SOAP / XMPP / etc.
  - Support for sessions
  - Master slave setup for distributed load testing

Distributed Tsung

- Although Tsung provided us most of 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
  - http://rightaws.rubyforge.org/
- Which uploads the results to S3 after each run



tsung - Statistics

Main Statistics

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.894 msec	0.298 msec	5703.5 / sec	0.330 msec	7647824
page	98.32 msec	2.10 msec	5703.3 / sec	2.22 msec	7647817
request	0.432 msec	0.283 msec	36952.6 / sec	0.307 msec	50280854
session	1200.4 sec	1200.0 sec	416.5 / sec	1200.0 sec	443059

Transactions Statistics

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
tr_login	0.101 sec	2.95 msec	690.4 / sec	3.40 msec	1098484

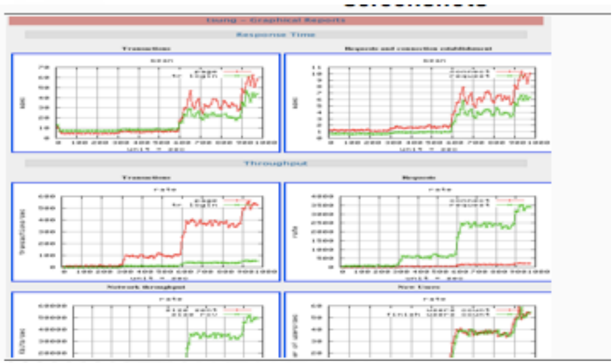
Network Throughput

Name	Highest Rate	Total
size_rcv	521.40 Mbits/sec	86.44 GB
size_sent	39.10 Mbits/sec	6.54 GB

Counters Statistics

Name	Highest Rate	Total number
finish_users_count	416.5 / sec	446996
newphase	2.1 / sec	612
users_count	681 / sec	1105176

Name	Max
connected	89288
users	658180



Performance

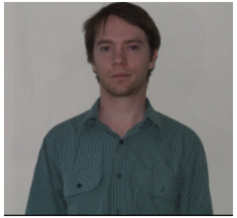
velti

- 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 of our engineers
  - Ben Murphy



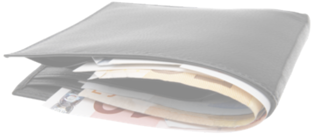
  - David Townsend

- Why not try it out for yourselves?

git@github.com:mitadmin/ionstorm.git

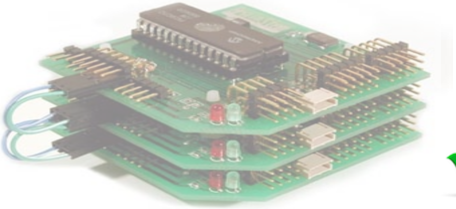
Battle Stories #2

- Building a wallet



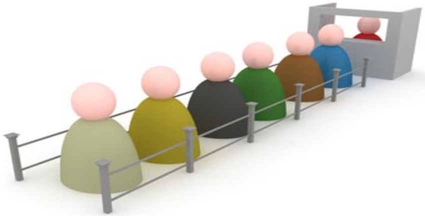
✓

- Optimizing your hardware stack



✓

- Building a robust queue – final version



Building a Queue

velti

- 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

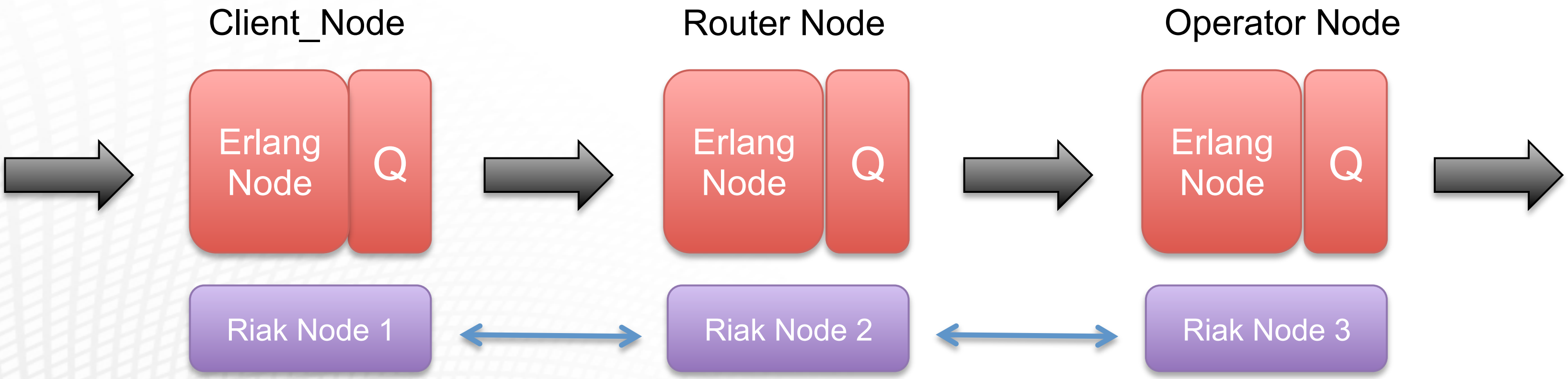
- With a little help from the Basho guys we came up with a new approach

- 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 : restart_count : counter “
  - **node:** the name of the originating node for the request e.g “client_node”
  - **date:** e.g. “2012-01-01”
  - **restart_count:** number of node restarts e.g. “2”
  - **counter:** number of message since last node restart or date change e.g. “3000”

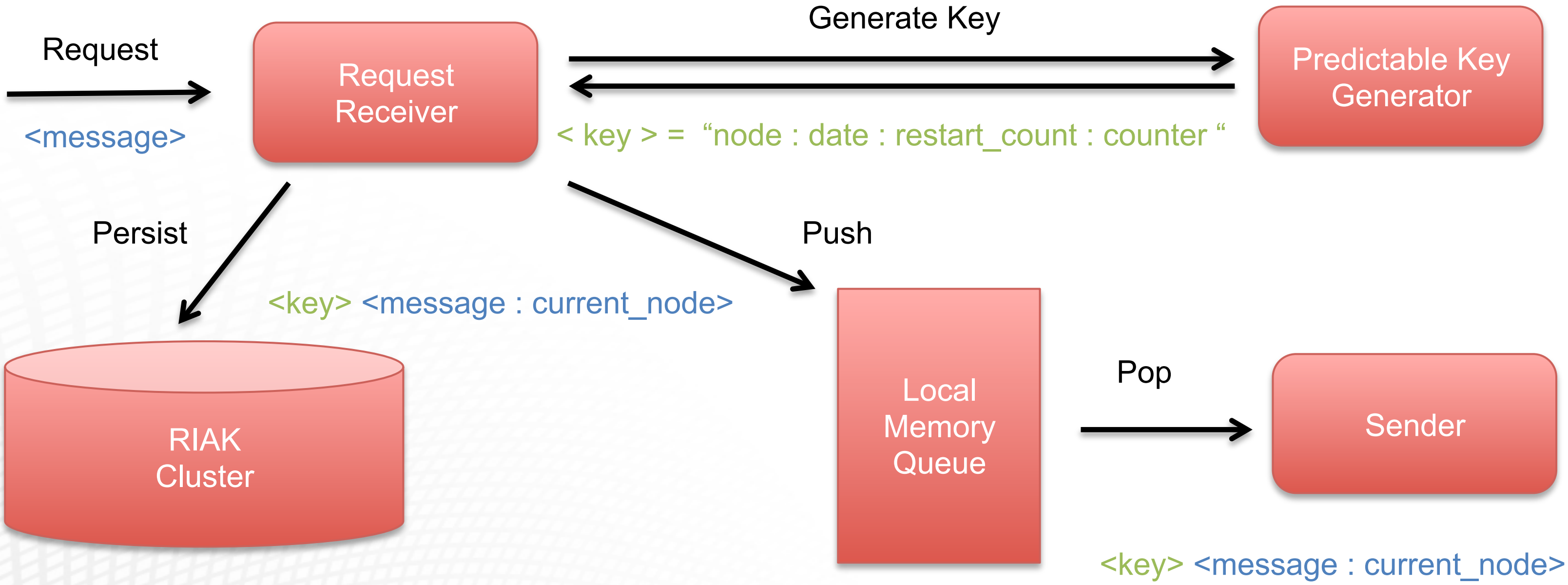
- Value: <message : current_node >
  - **message:** the original request e.g. “send sms”
  - **current_node:** the current node the message is located e.g. “router_node”

Snapshot

- Every 1000 messages
- Take a snapshot of the counter
  - Key: “ client_node : 2012-01-01 : 1 : 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



Request

<message>

Request Receiver

Generate Key

Predictable Key Generator

< key > = "node : date : restart_count : counter "

Persist

Push

<key> <message : current_node>

RIAK Cluster

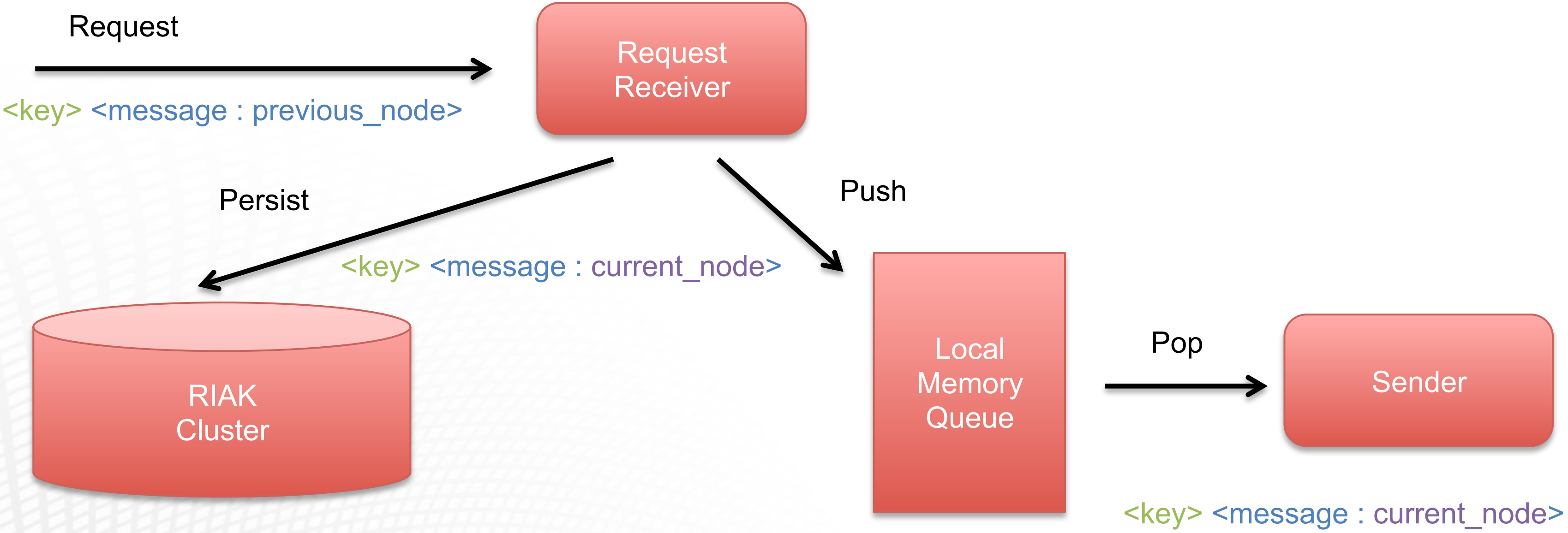
Local Memory Queue

Pop

Sender

<key> <message : current_node>

Queue – intermediate node



Request

<key> <message : previous_node>

Request Receiver

Persist

Push

<key> <message : current_node>

RIAK Cluster

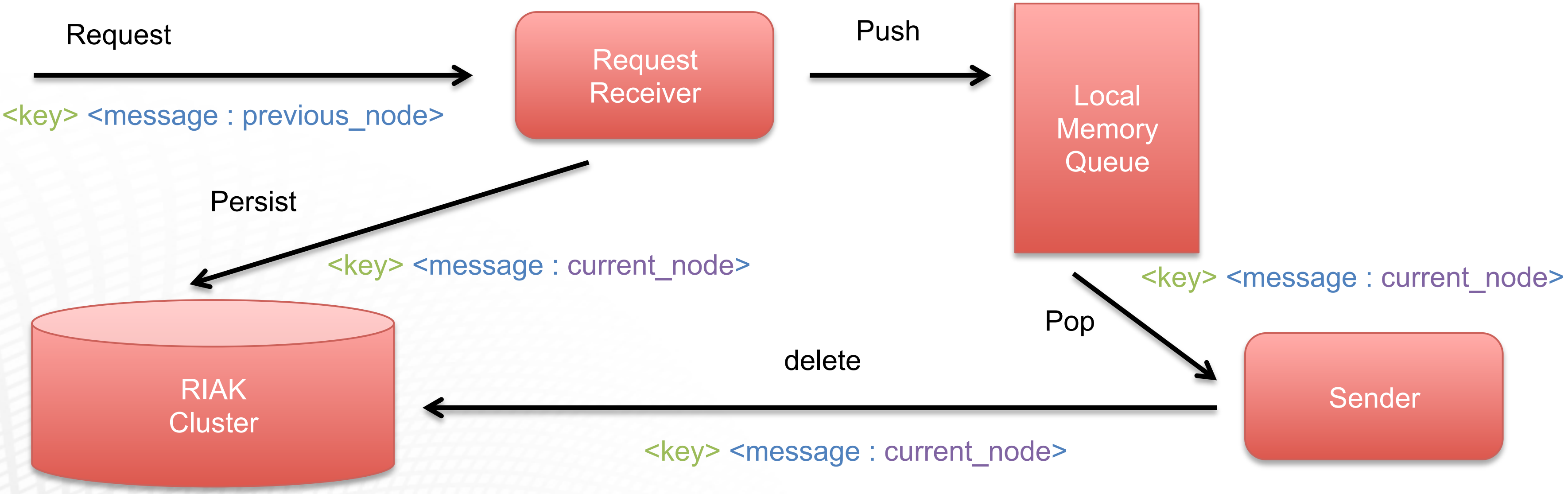
Local Memory Queue

Pop

Sender

<key> <message : current_node>

Queue – outgoing node


Request
<key> <message : previous_node>
Request Receiver
Push
Local Memory Queue
Persist
<key> <message : current_node>
<key> <message : current_node>
Pop
delete
RIAK Cluster
Sender
<key> <message : current_node>

Recovery

- Identify node that needs recovery e.g. “client_node”
- Take the current date e.g. “2012-01-01”
- Request from RIAK the current restart_count e.g. “1”
- Use the snapshot to get the last current count recorded e.g. “3000”
  - Key: “ client_node : 2012-01-01 : 1 : snapshot ”
  - Value: 3000
- Create a temporary recovery node
- Rebuild by walking the keys from:
  - from the value: 1
  - to the current count + (2 x snapshot interval): 5000
- Once complete create the original node & discard the recovery node

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

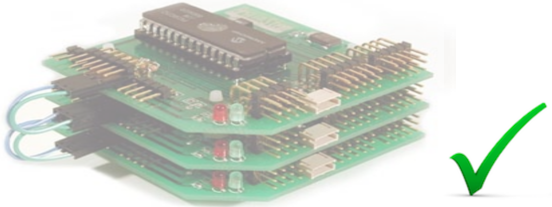
velti

- 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 soloution

Battle Stories #2

- Building a wallet

- Optimizing your hardware stack



- Building a robust queue

Thank You

velti

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