Social Networks and the Richness of Data

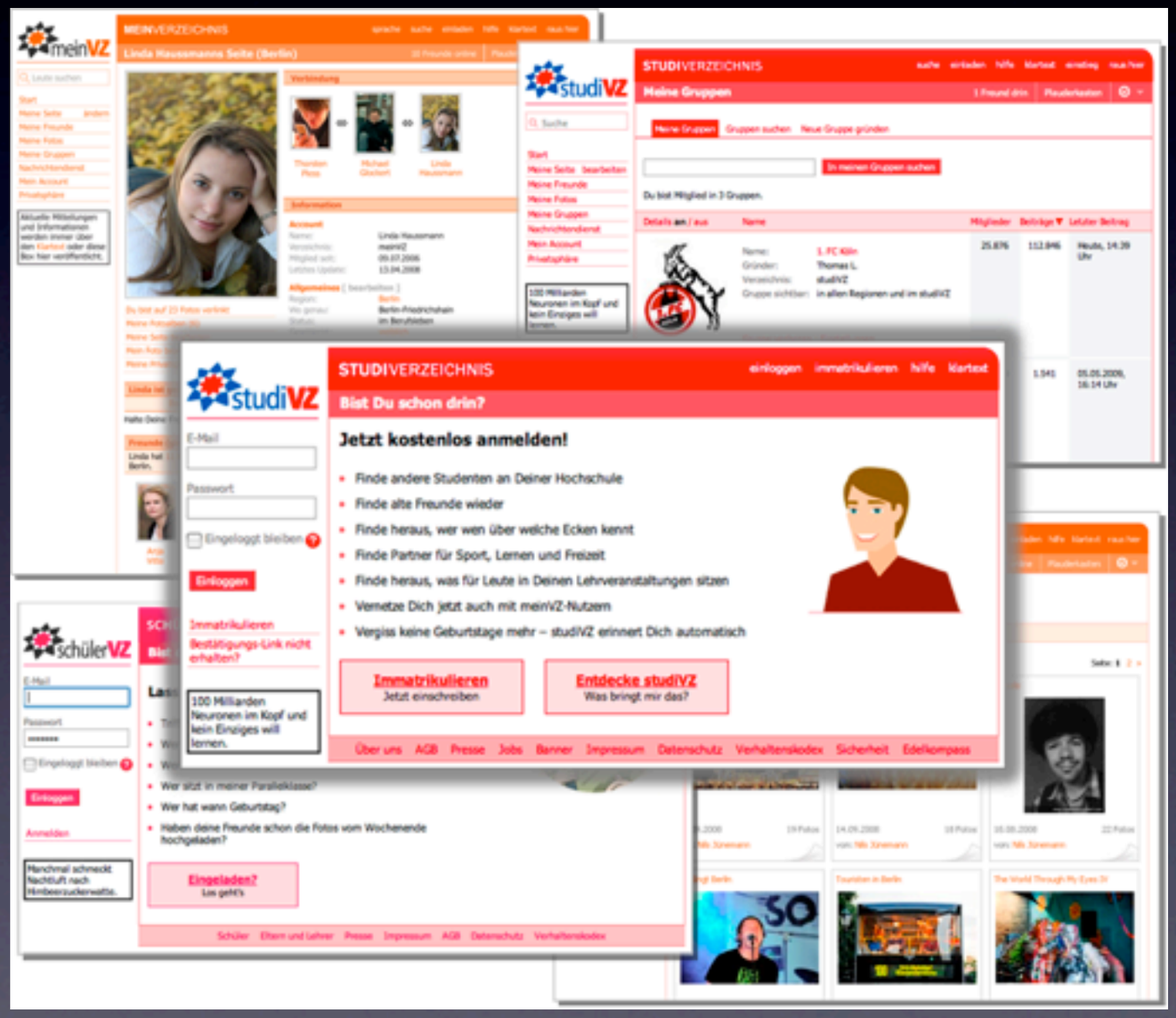
Getting distributed Webservices Done with NoSQL

Fabrizio Schmidt, Lars George
VZnet Netzwerke Ltd.

Content

- Unique Challenges
- System Evolution
- Architecture
- Activity Stream - NoSQL
- Lessons learned, Future

Unique Challenges



- 16 Million Users
- > 80% Active/Month
- > 40% Active/Daily
- > 30min Daily Time on Site

Germany, December 2009

	Unique Users Mio./Month	**Stickiness*** Percent	**Visits** Mio./Month	**Time Spent on Site** Mio. Minutes/Month
T-Online**	21,7	19,1	226,4	775
VZ gesamt	16,4	39,4	458,2	3.726
Facebook	10,6	25,0	191,0	828
Wer-kennt-wen	7,0	34,6	157,0	1.364
StayFriends	5,7	10,8	23,8	85
MySpace	5,6	14,4	42,7	202
Xing	3,7	14,1	22,1	84
Lokalisten	2,0	31,3	40,4	356
Netlog	1,6	18,3	15,7	116

Source: Comscore
100120 Comscore Dez 2009

* Share of daily unique users
** Not a social network, for comparison only

schülerVZ studiVZ meinVZ

Unique Challenges

- 16 Million Users
- 1 Billion Relationships
- 3 Billion Photos
- 150 TB Data

- 13 Million Messages per Day
- 17 Million Logins per Day
- 15 Billion Requests per Month
- 120 Million Emails per Week

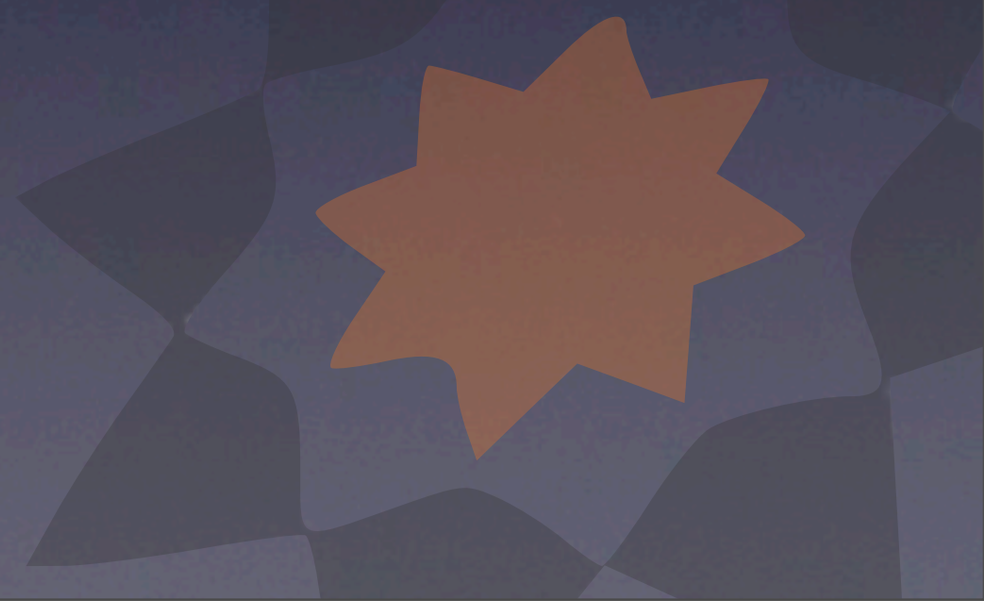
Old System - Phoenix

- LAMP
- Apache + PHP + APC (50 req/s)
- Sharded MySQL Multi-Master Setup
- Memcache with 1 TB+

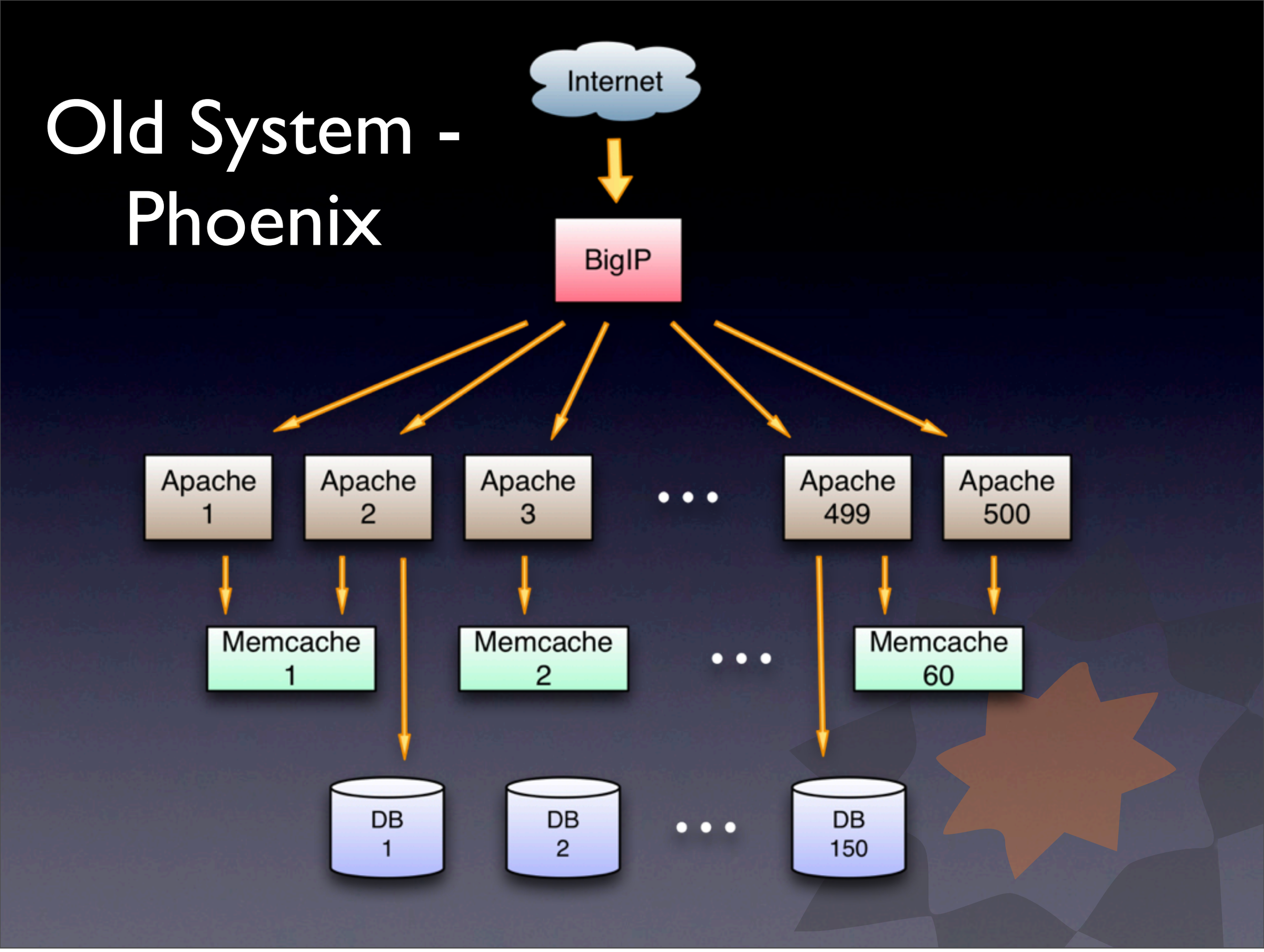
Monolithic Single Service, Synchronous

Old System - Phoenix

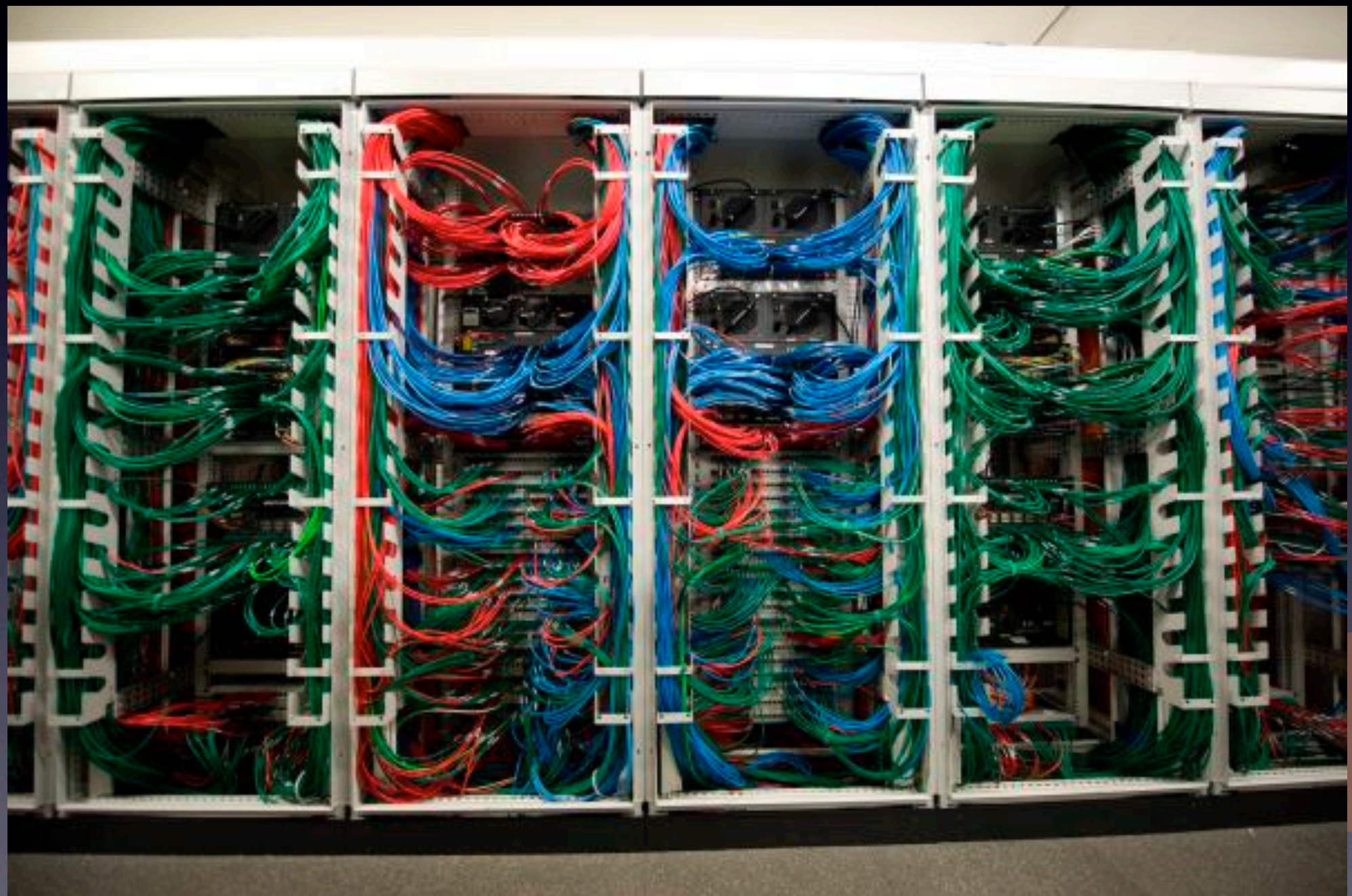
- 500+ Apache Frontends
- 60+ Memcaches
- 150+ MySQL Servers



Old System - Phoenix



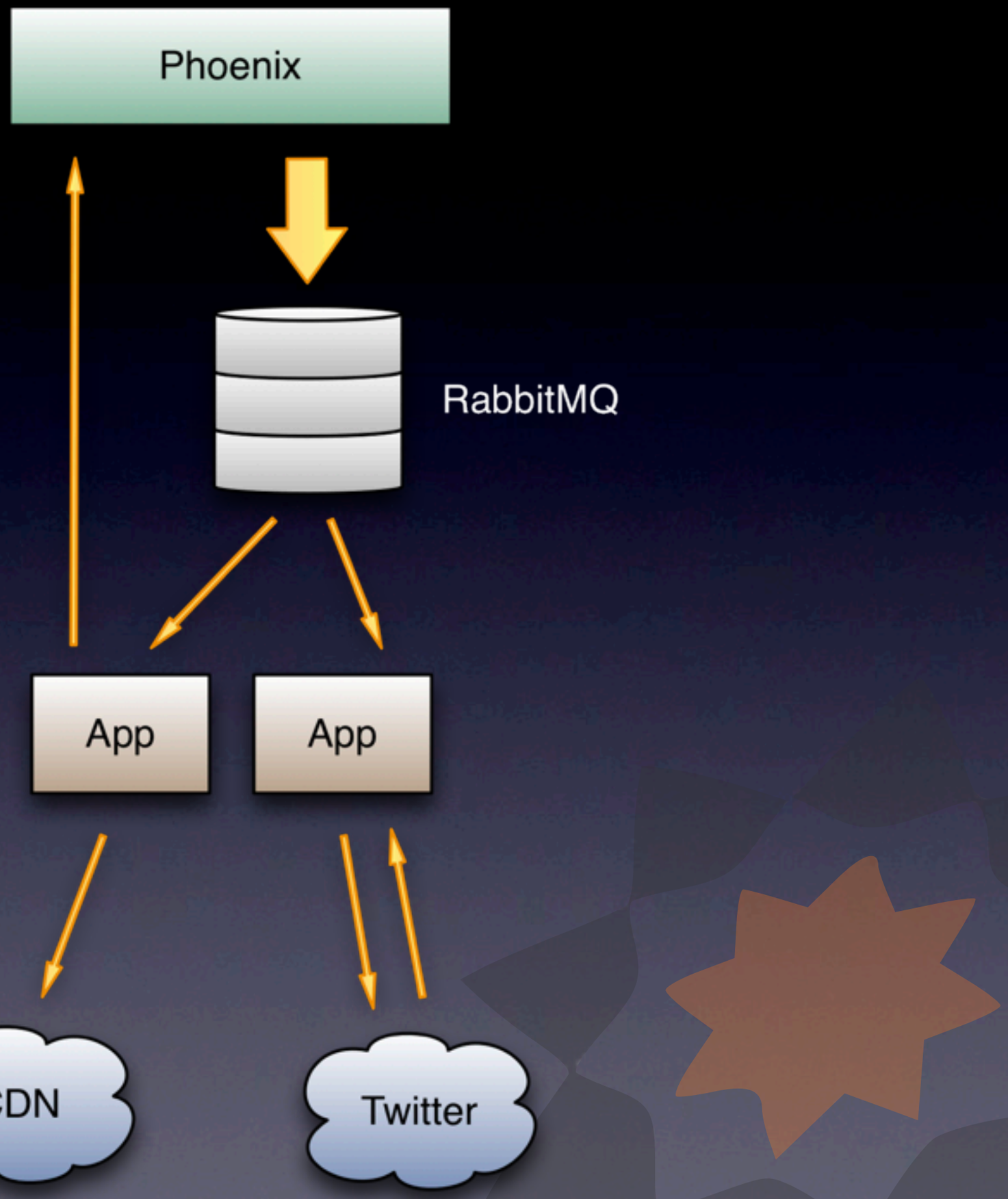
DON‘T PANIC



Asynchronous Services

- Basic Services
  - Twitter
  - Mobile
  - CDN Purge
  - ...
- Java (e.g. Tomcat)
- RabbitMQ

First Services



Phoenix - RabbitMQ

1. PHP Implementation of AMQP Client

   Too slow!

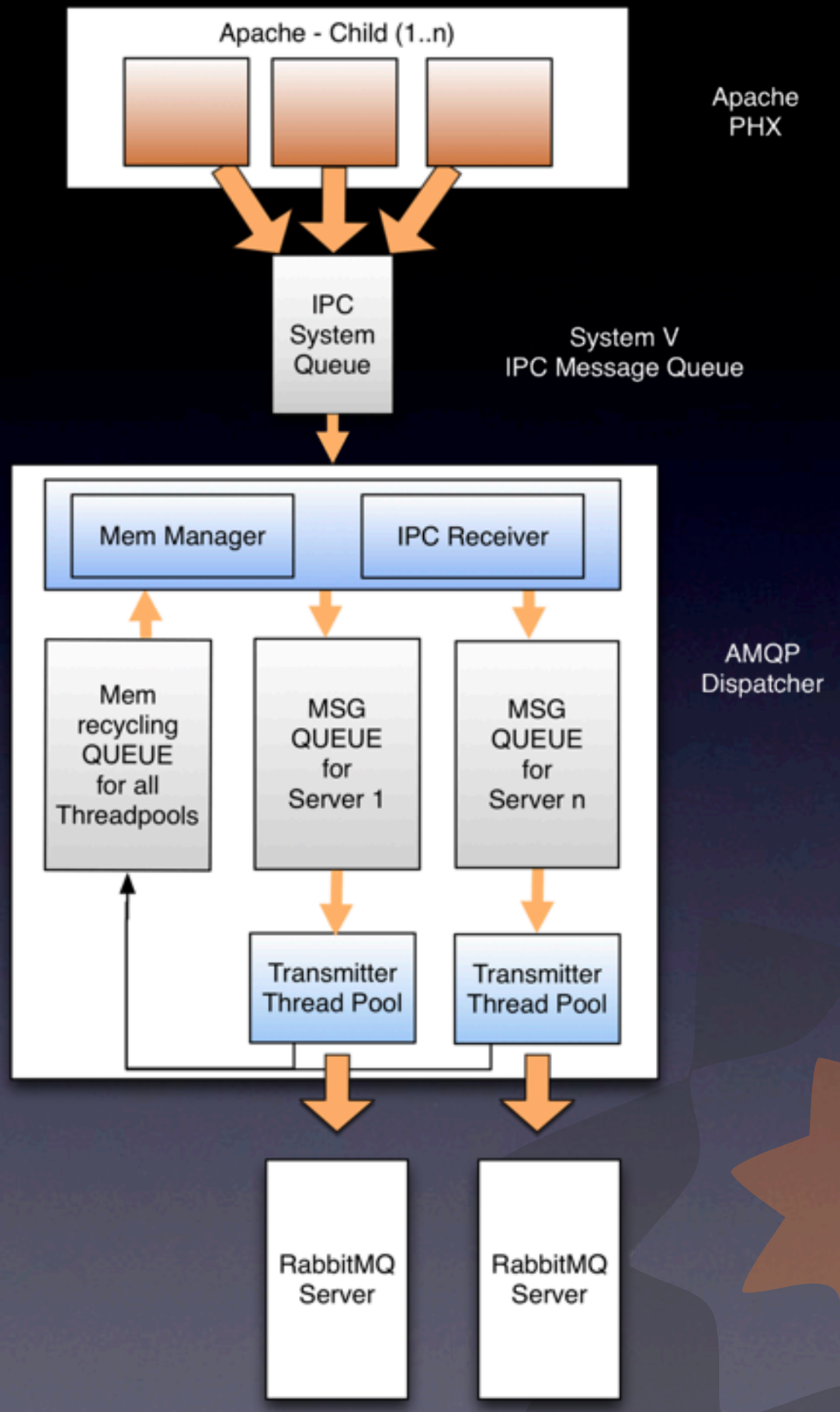
2. PHP C - Extension (php-amqp http://code.google.com/p/php-amqp/)

   Fast enough

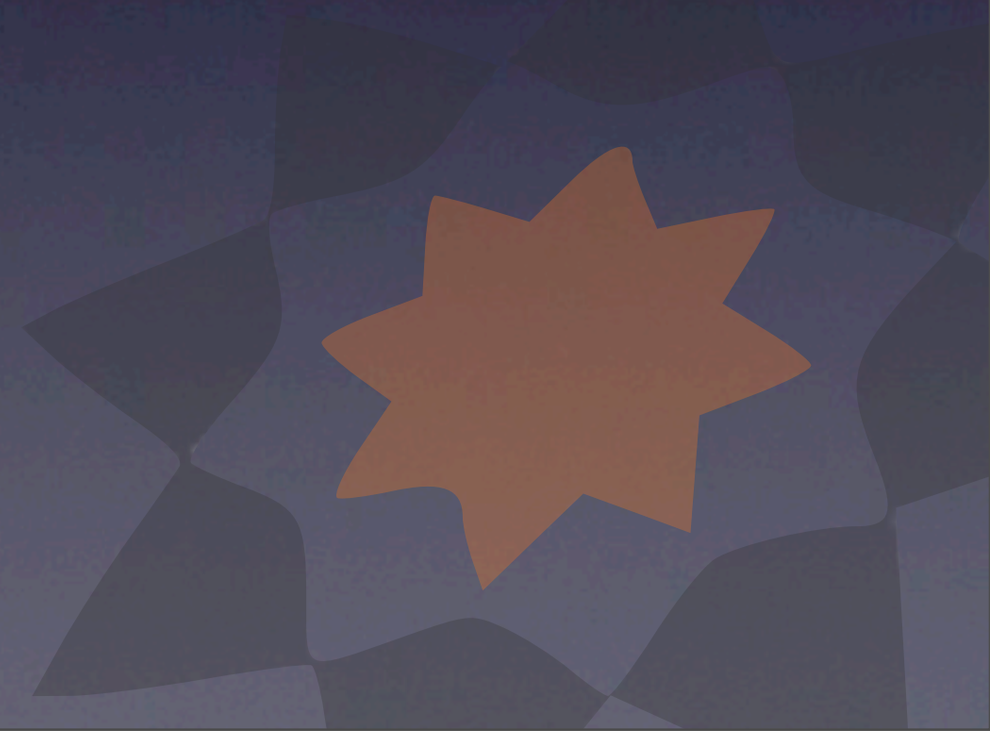
3. IPC - AMQP Dispatcher C-Daemon

   That's it! But not released so far

IPC - AMQP Dispatcher



Activity Stream



Old Activity Stream

- Memcache only - no persistence
- Status updates only
- #fail on users with >1000 friends
- #fail on memcache restart

Old Activity Stream

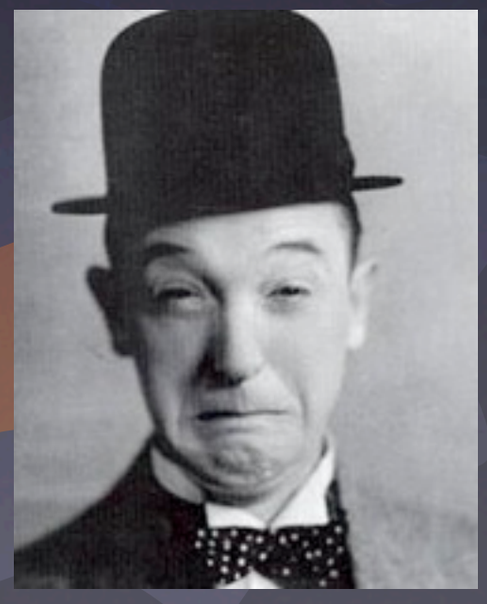
We cheated!

- Memcache only - no persistence
- Status updates only
- #fail on users with >1000 friends
- #fail on memcache restart

Old Activity Stream

We cheated!

- Memcache only - no persistence
- Status updates only
- #fail on users with >1000 friends
- #fail on memcache restart



source: internet

Social Network Problem

= Twitter Problem???

- >15 different Events
- Timelines
- Aggregation
- Filters
- Privacy

Do the Math!

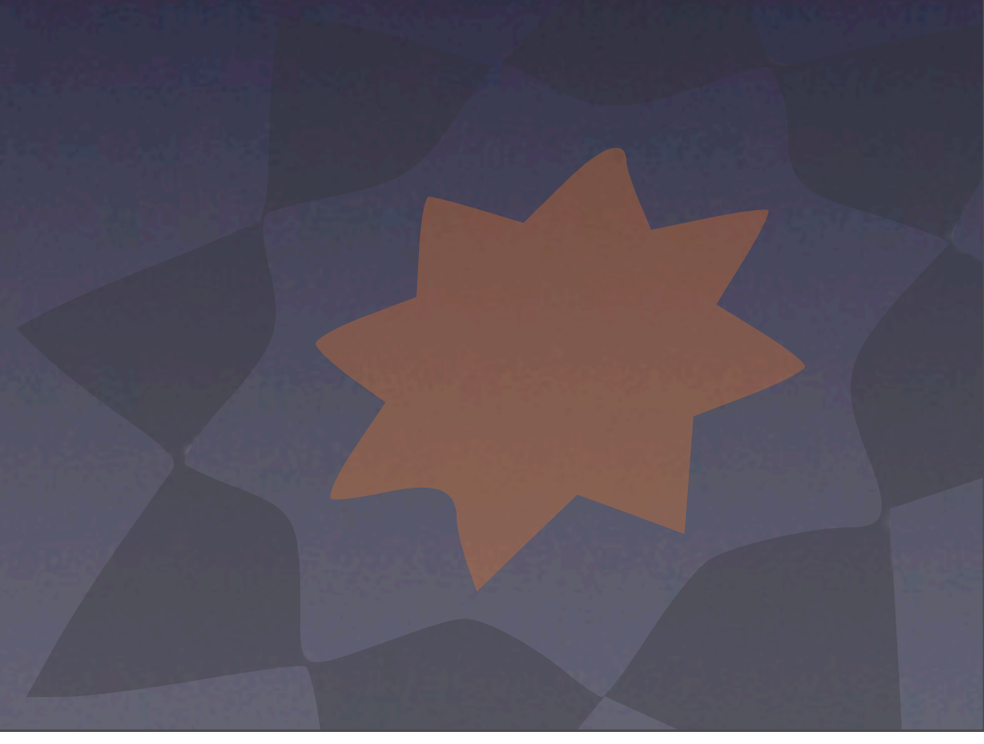
Do the Math!

18M Events/day sent to ~150 friends

Do the Math!

18M Events/day sent to ~150 friends

=> 2700M timeline inserts / day

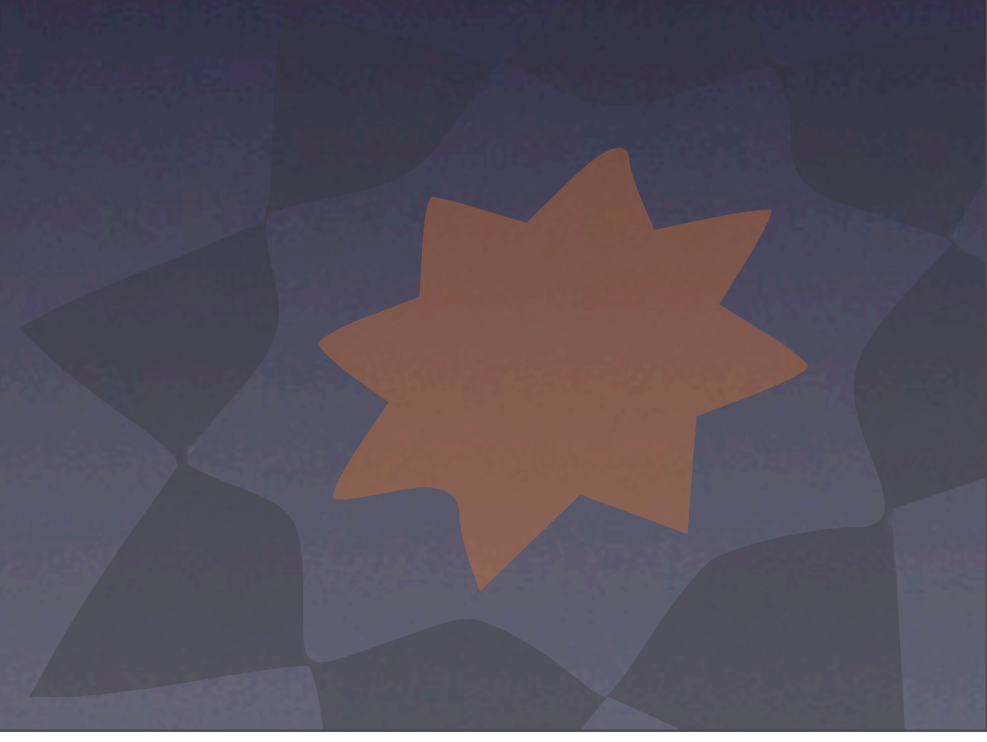


Do the Math!

18M Events/day sent to ~150 friends

=> 2700M timeline inserts / day

20% during peak hour



Do the Math!

18M Events/day sent to ~150 friends

=> 2700M timeline inserts / day

20% during peak hour

=> 3.6M event inserts/hour - 1000/s

Do the Math!

18M Events/day sent to ~150 friends

=> 2700M timeline inserts / day

20% during peak hour

=> 3.6M event inserts/hour - 1000/s

=> 540M timeline inserts/hour - 150000/s

e inserts / day

r

erts/hour - 1000/s

inserts/hour - 150000/s

New Activity Stream

- Social Network Problem
- Architecture
- NoSQL Systems

New Activity Stream

Do it right!

- Social Network Problem
- Architecture
- NoSQL Systems

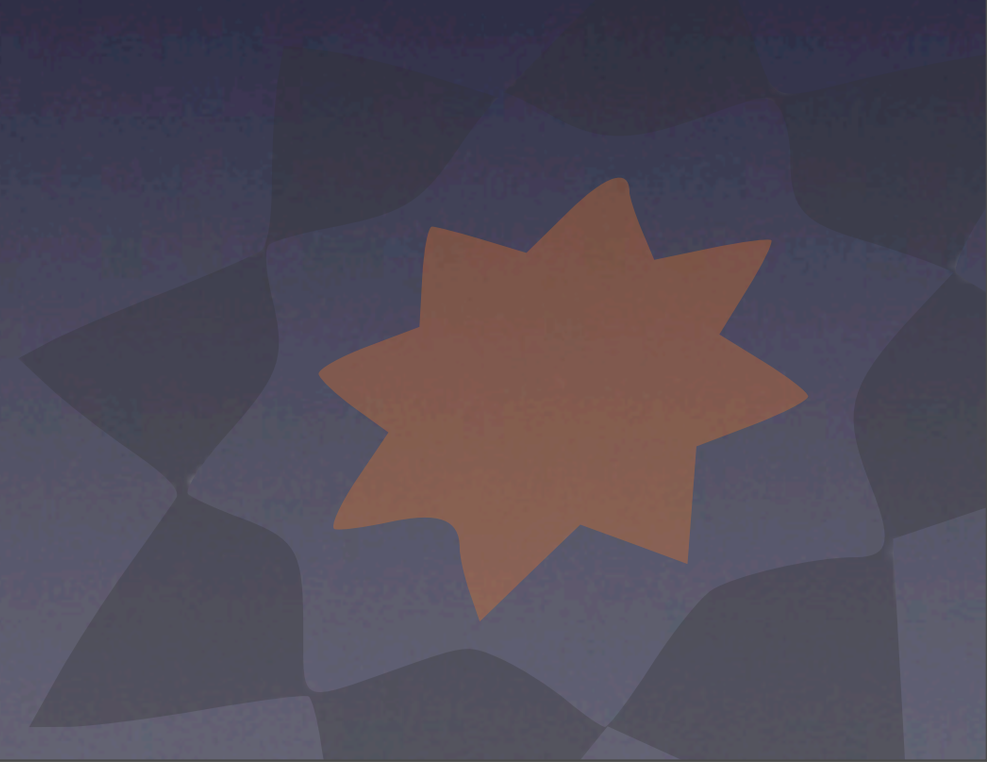
New Activity Stream

Do it right!

- Social Network Problem
- Architecture
- NoSQL Systems

source: internet

Architecture



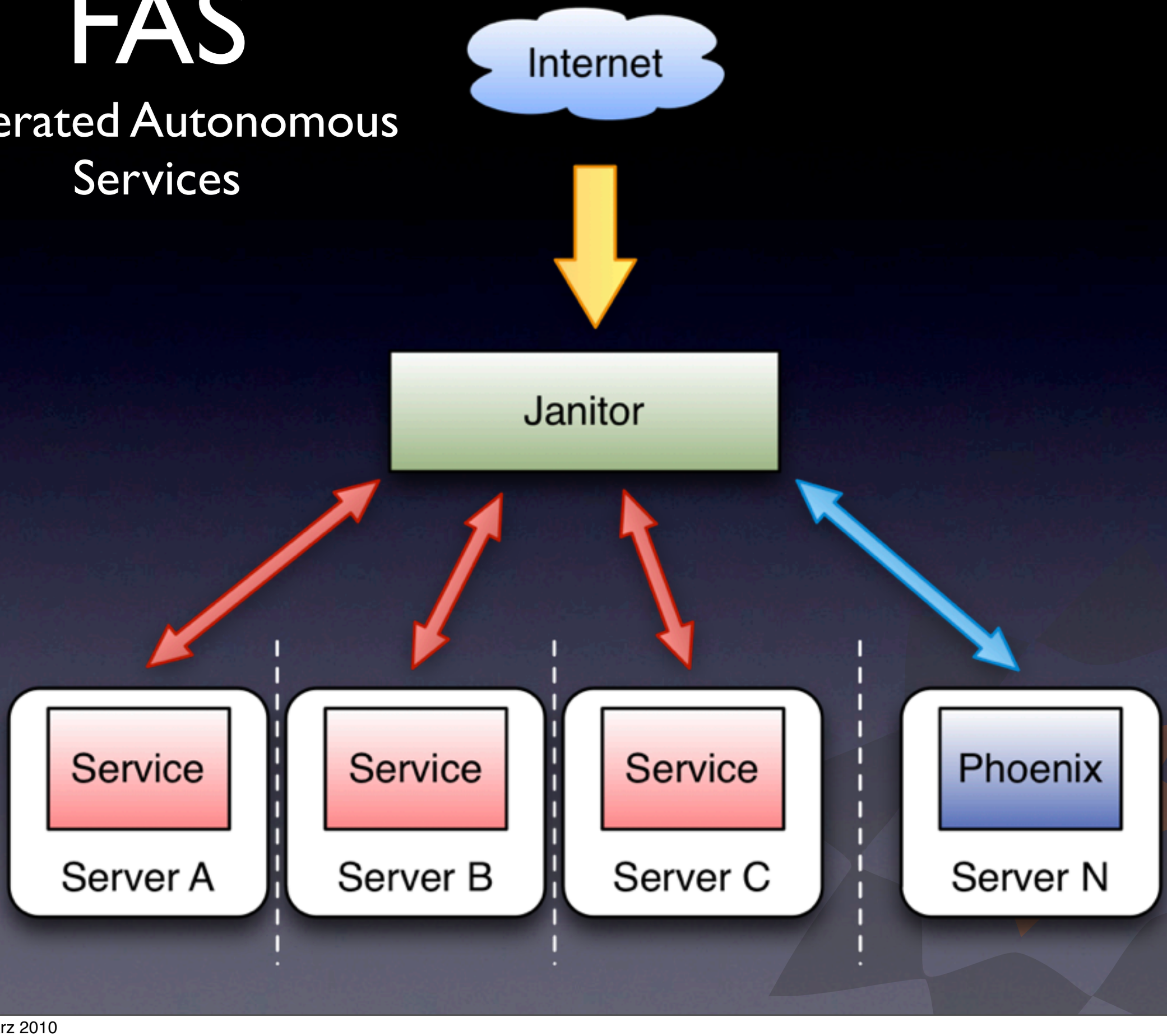
FAS

Federated Autonomous Services

- Nginx + Janitor
- Embedded Jetty + RESTeasy
- NoSQL Storage Backends

FAS

Federated Autonomous Services



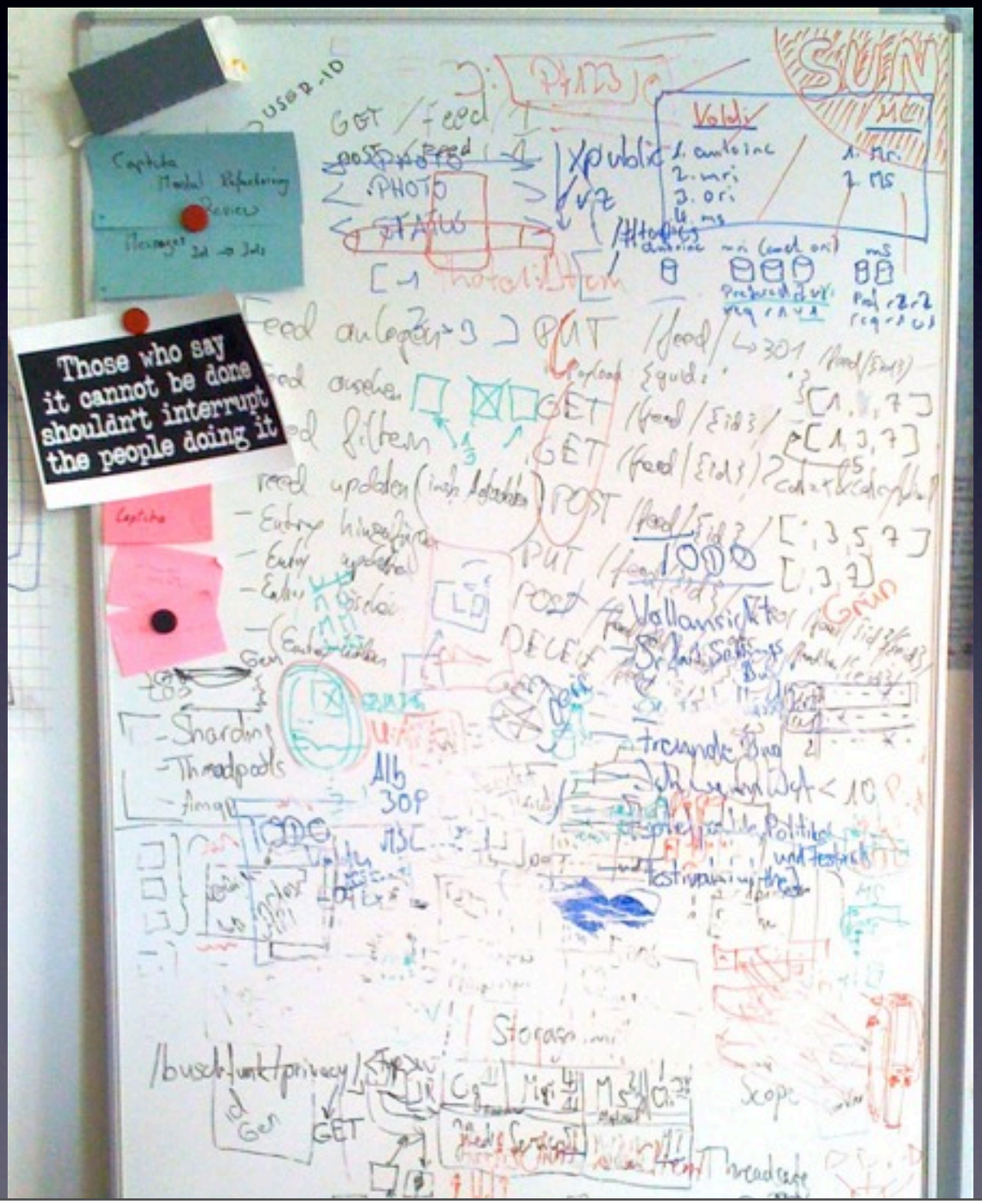
Activity Stream

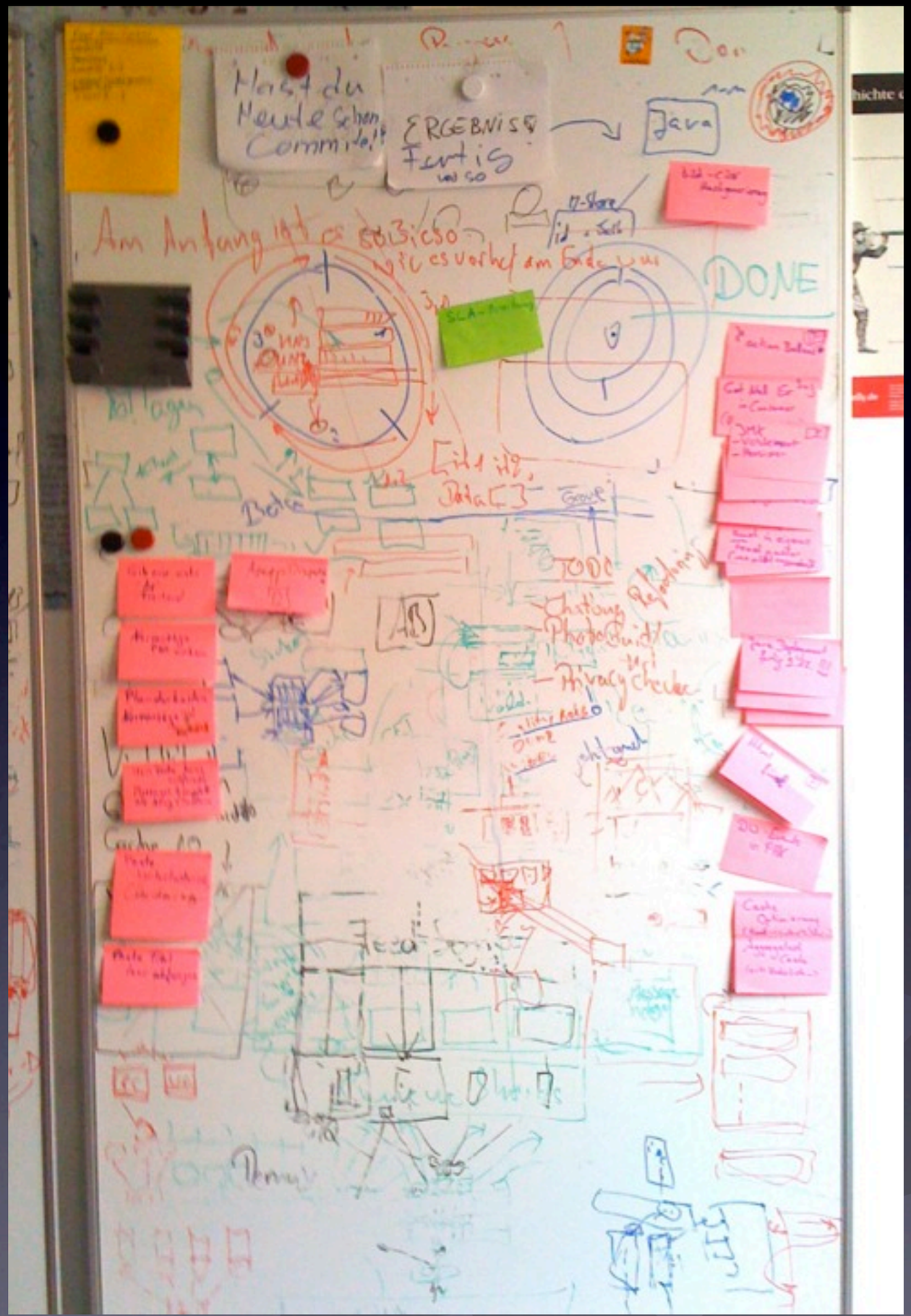
as a service

Requirements:

- Endless scalability
- Storage & cloud independent
- Fast
- Flexible & extensible data model

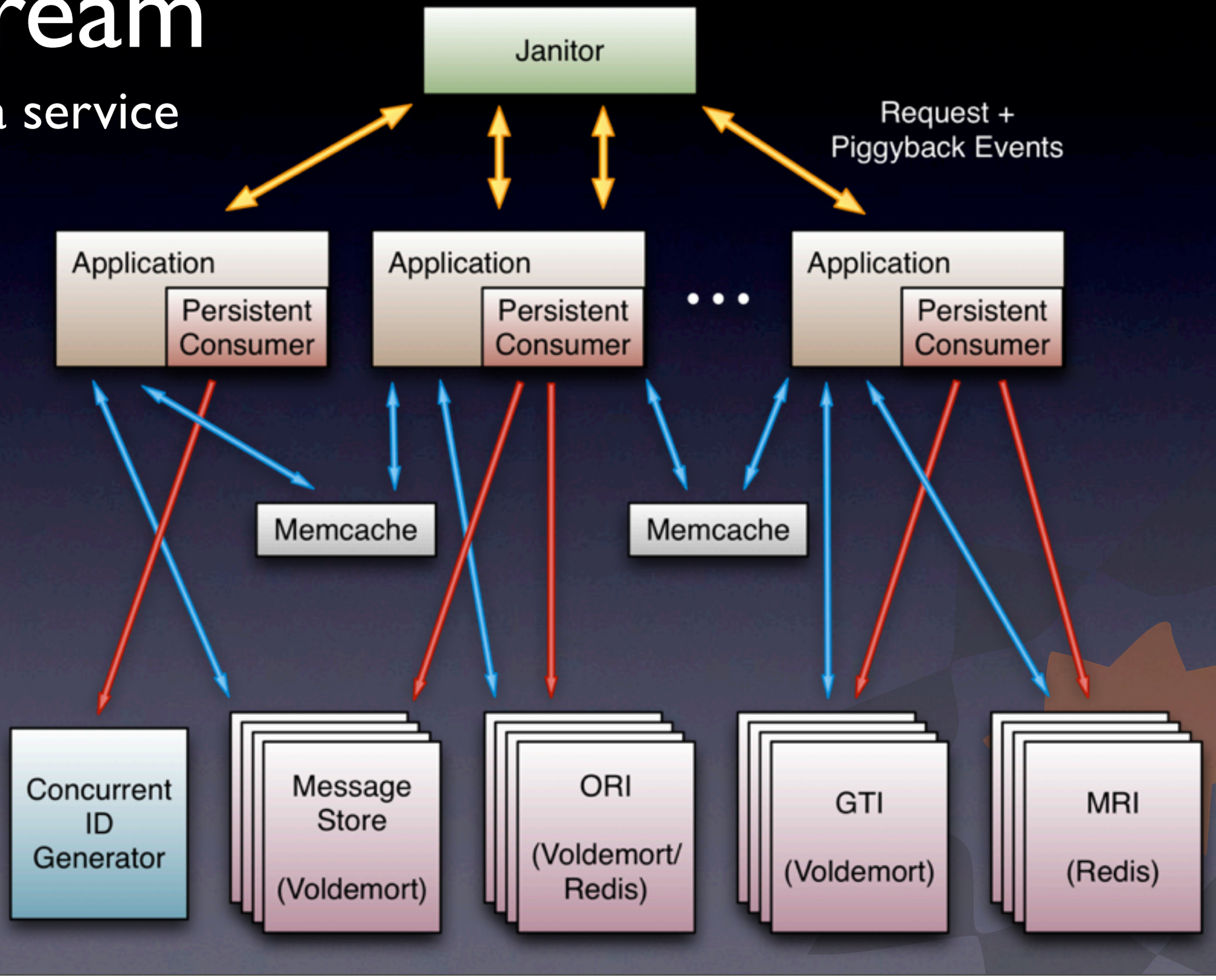
Thinking in layers...





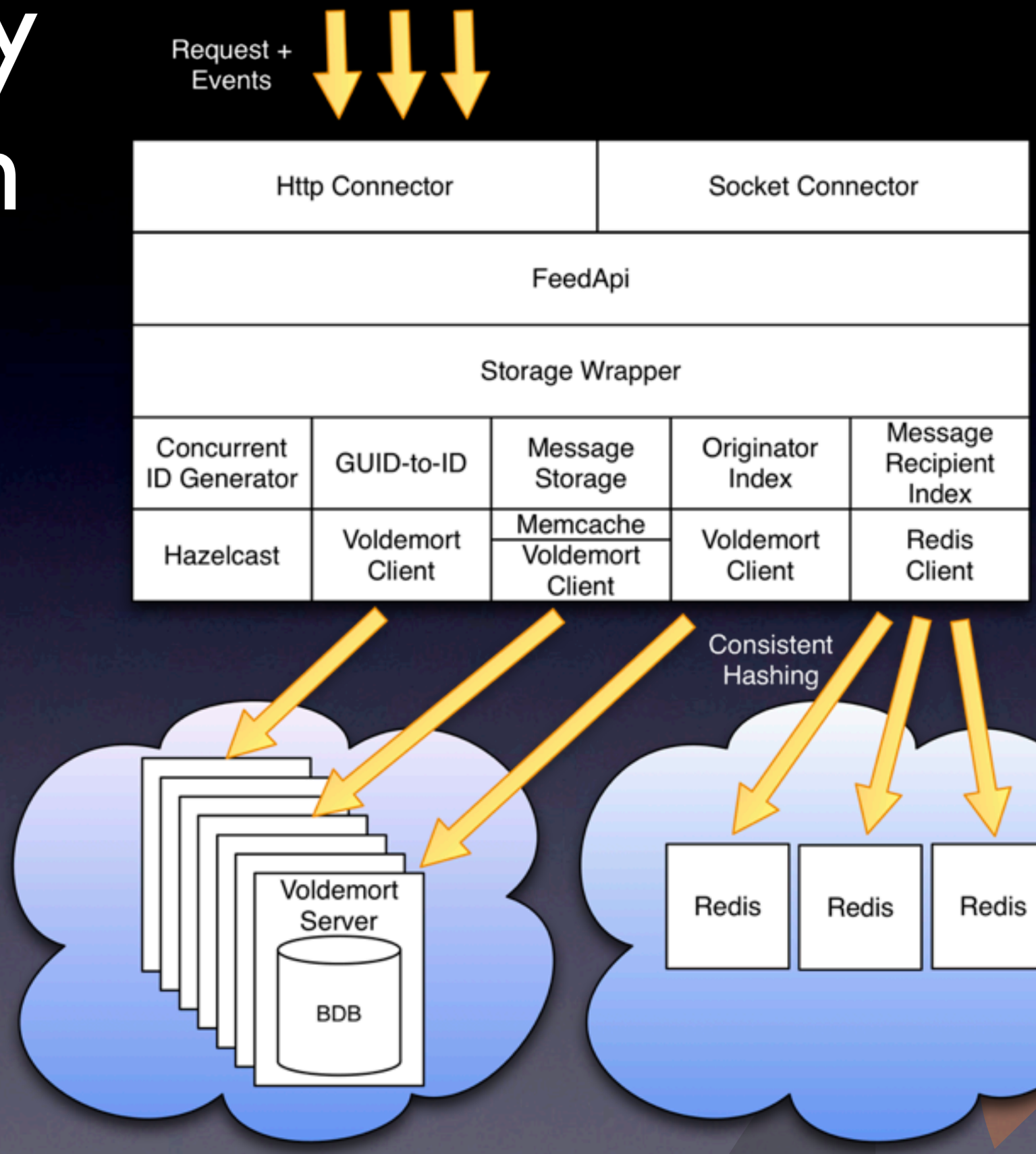
Activity Stream

as a service

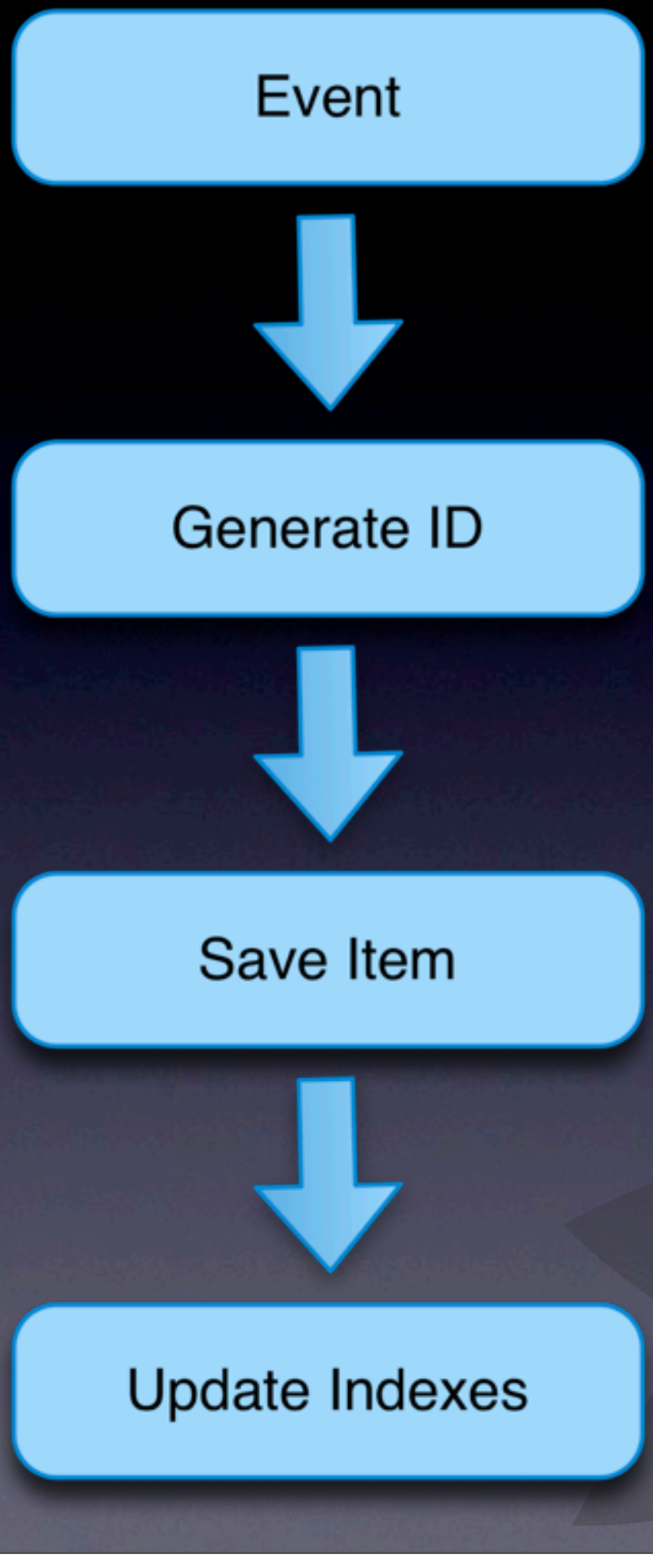


Activity Stream

as a service



NoSQL Schema



Event

Event is sent in by piggybacking the request

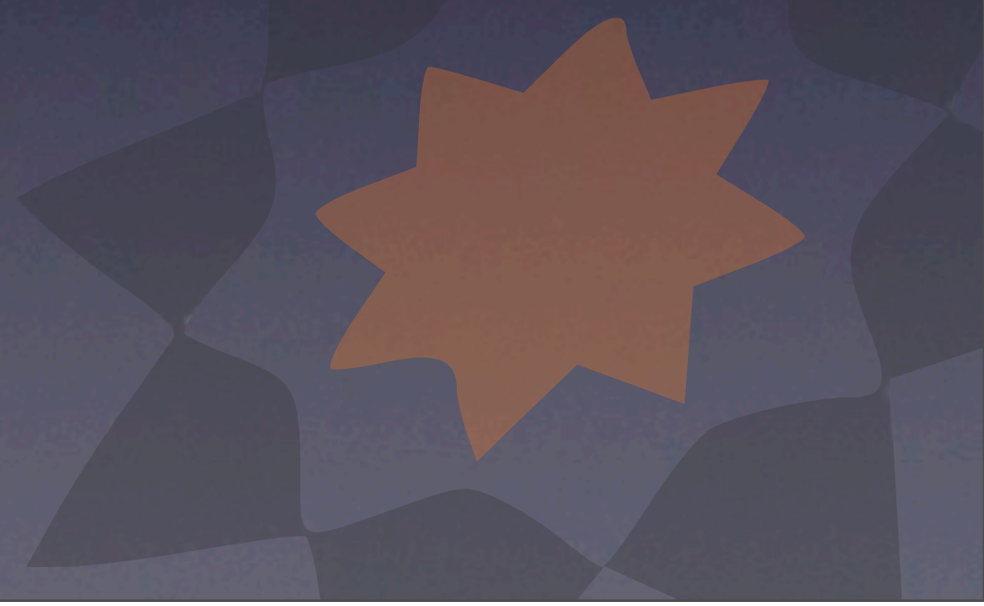
Generate ID

Event

Generate ID

Generate itemID - unique ID of the event

Save Item

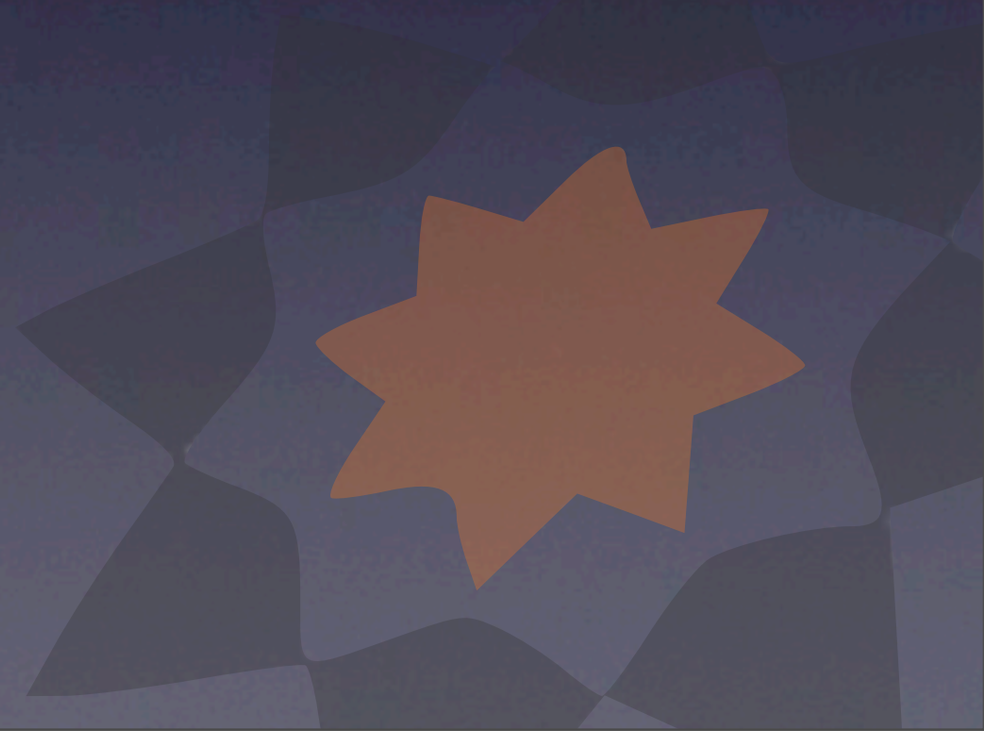


Generate ID

Save Item

itemID => stream_entry - save the event with meta information

Update Indexes



Save Item

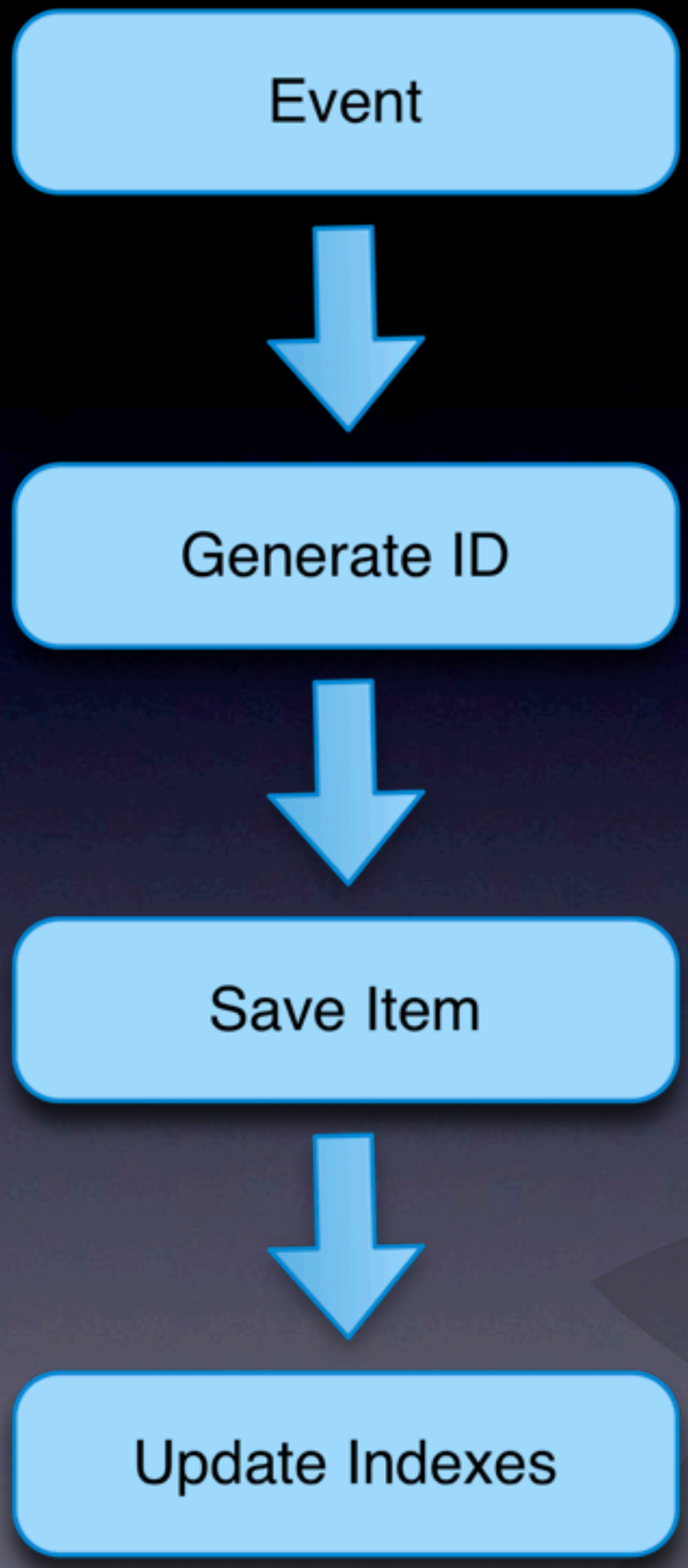
Update Indexes

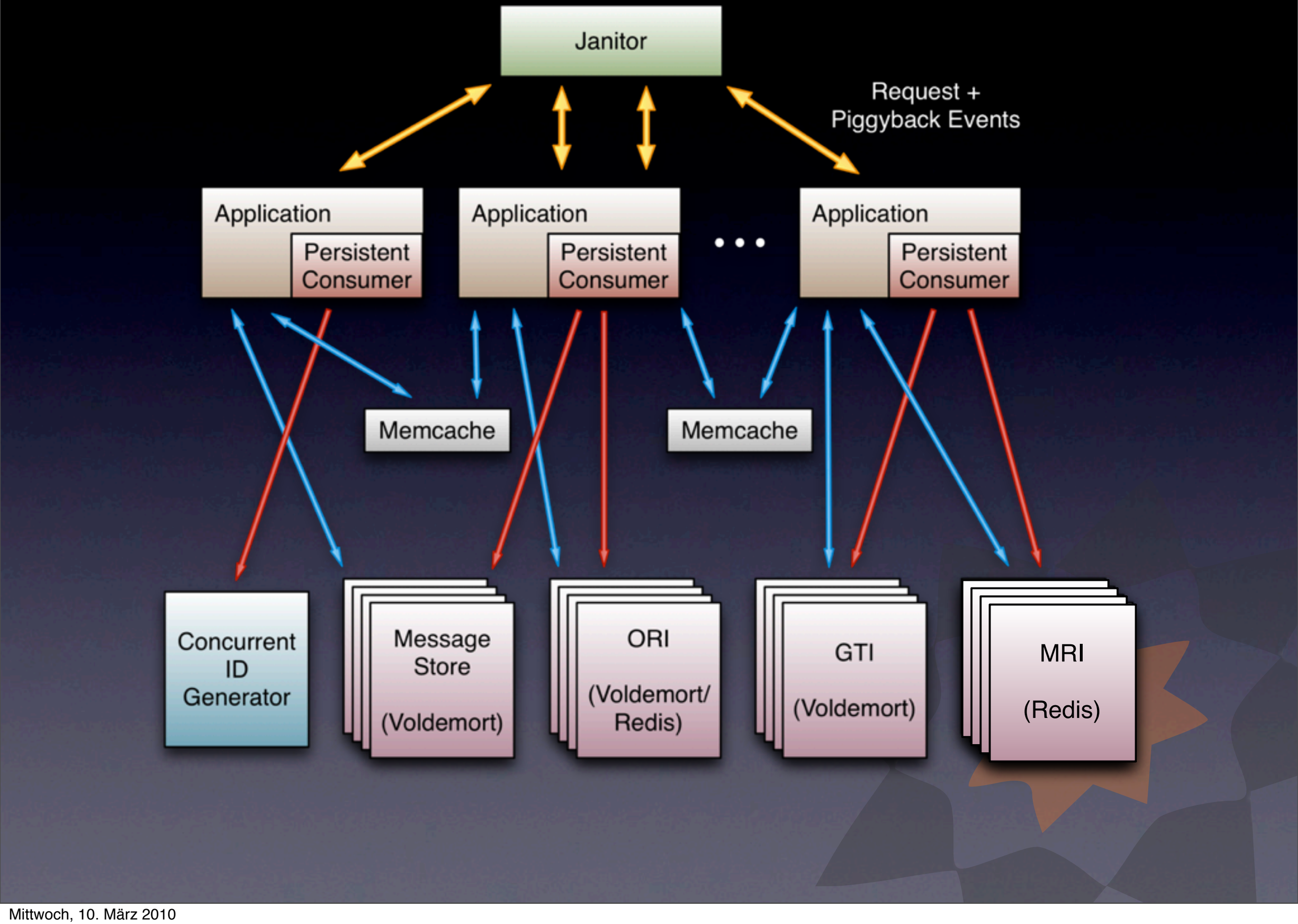
Insert into the timeline of each recipient

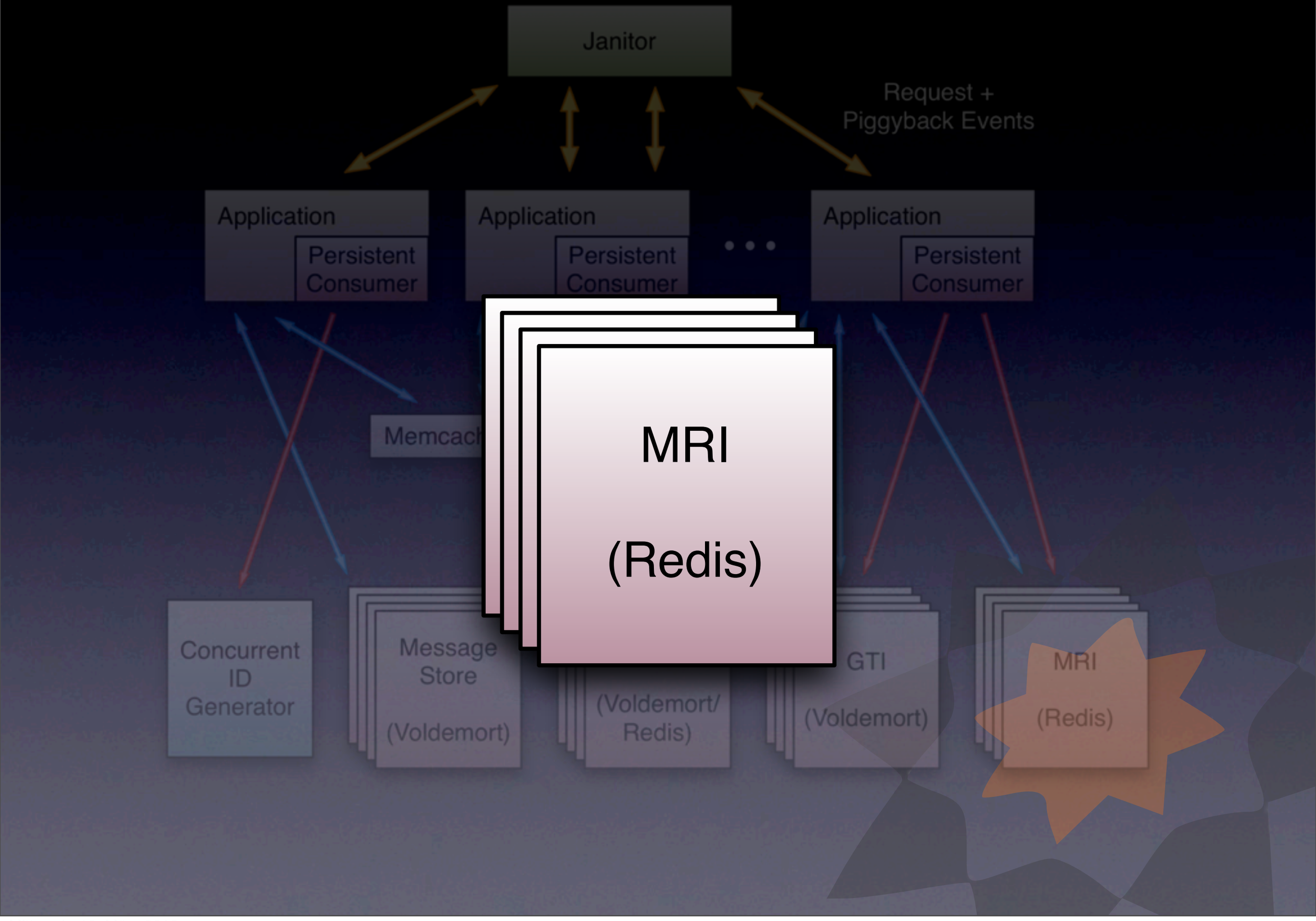
recipient → [[itemId, time, type], …]

Insert into the timeline of the event originator

sender → [[itemId, time, type], …]


Event
Generate ID
Save Item
Update Indexes


Janitor
Request +
Piggyback Events
Application
Persistent
Consumer
Application
Persistent
Consumer
Application
Persistent
Consumer
Memcache
Memcache
Concurrent
ID
Generator
Message
Store
(Voldemort)
ORI
(Voldemort/
Redis)
GTI
(Voldemort)
MRI
(Redis)


Janitor
Request +
Piggyback Events
Application
Persistent
Consumer
Application
Persistent
Consumer
Application
Persistent
Consumer
Memcach
MRI
(Redis)
Concurrent
ID
Generator
Message
Store
(Voldemort)
(Voldemort/
Redis)
GTI
(Voldemort)
MRI
(Redis)

Architecture: Push

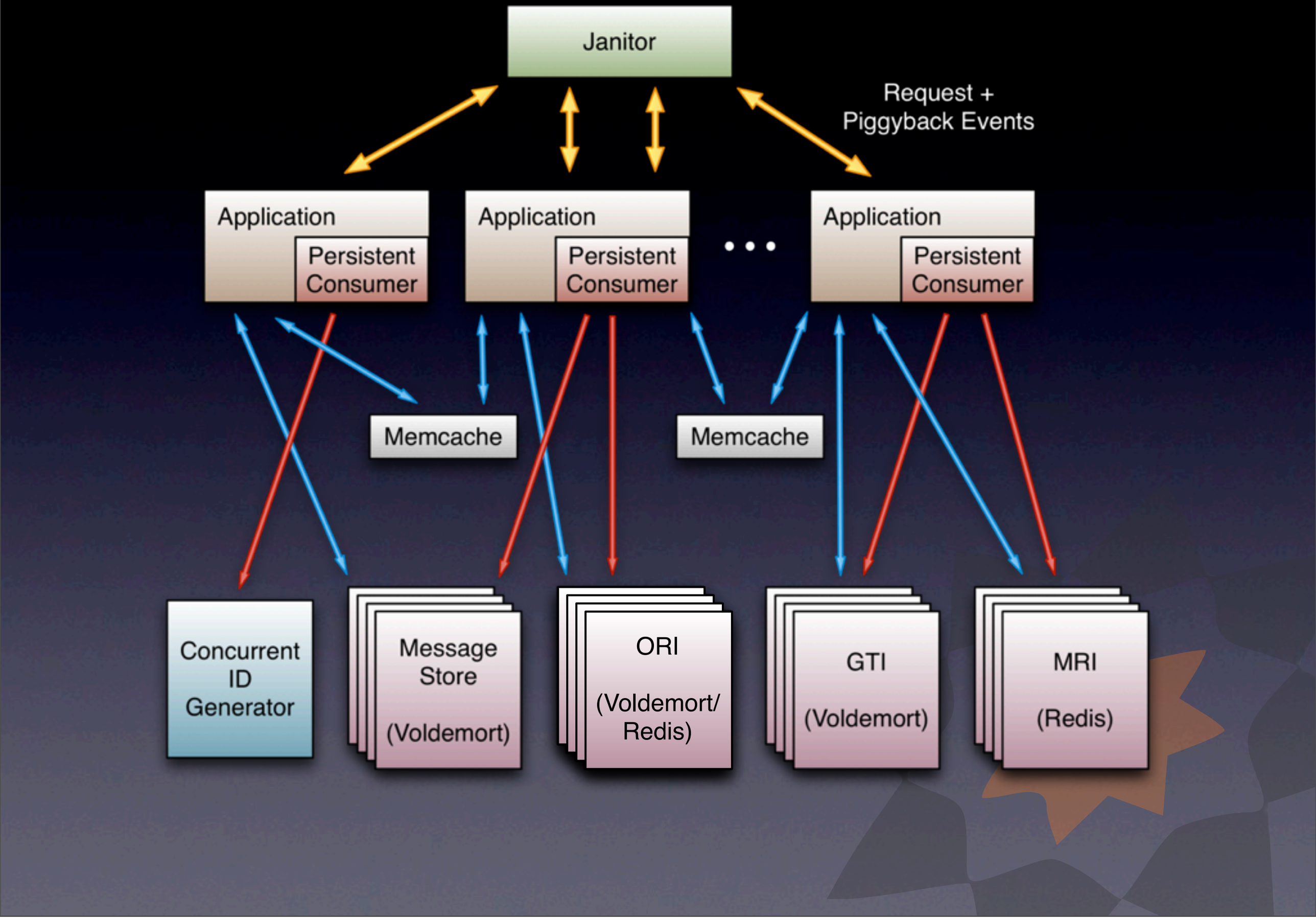
Message Recipient Index (MRI)

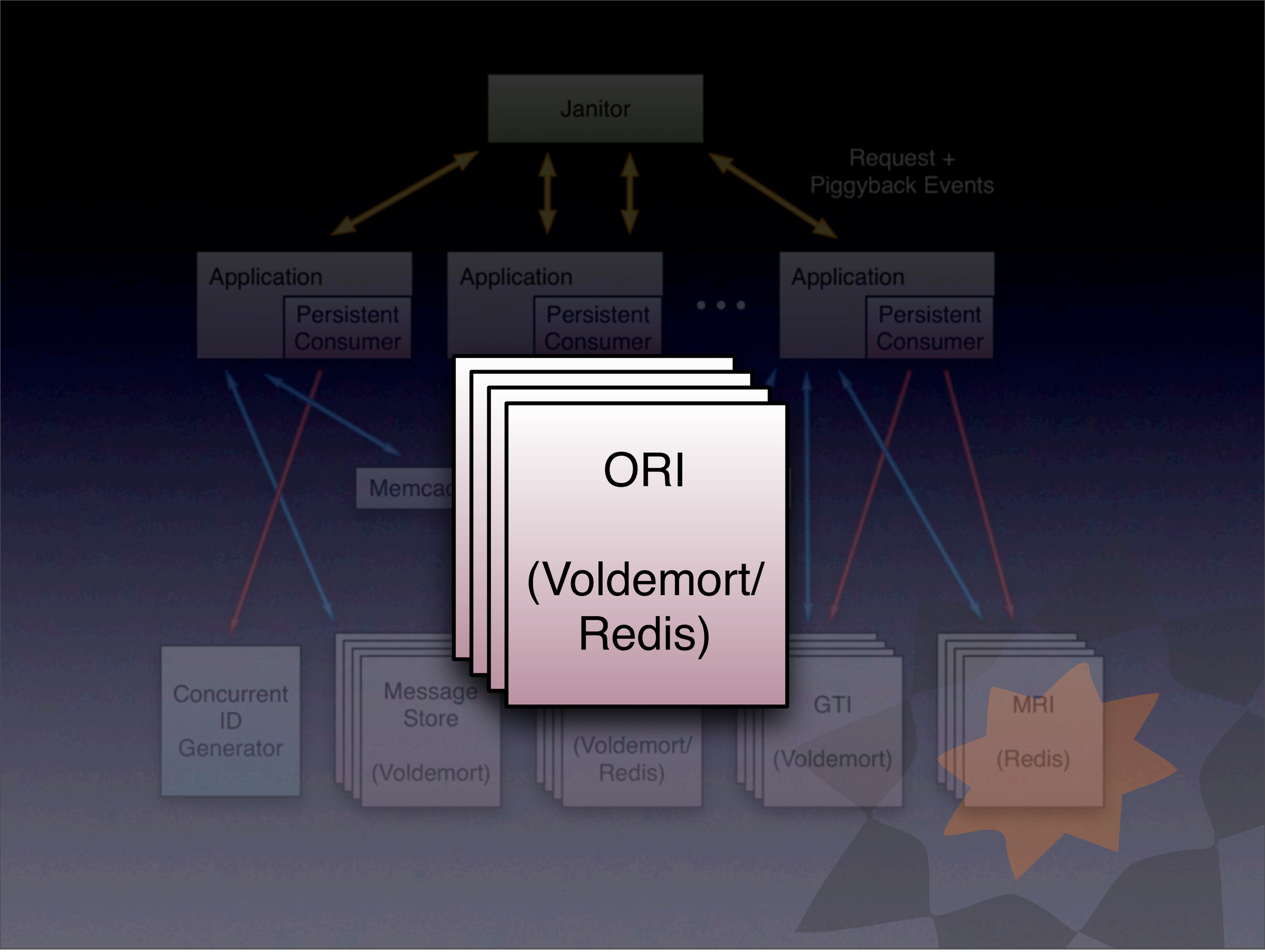
Push the Message directly to all MRIs

➡ {number of Recipients ~150} updates

Special profiles and some users have >500 recipients

➡ >500 pushes to recipient timelines => stress the system!


Janitor
Request +
Piggyback Events
Application
Persistent
Consumer
Application
Persistent
Consumer
Application
Persistent
Consumer
Memcache
Memcache
Concurrent
ID
Generator
Message
Store
(Voldemort)
ORI
(Voldemort/
Redis)
GTI
(Voldemort)
MRI
(Redis)


Janitor
Request +
Piggyback Events
Application
Persistent
Consumer
Application
Persistent
Consumer
Application
Persistent
Consumer
ORI
(Voldemort/
Redis)
Concurrent
ID
Generator
Message
Store
(Voldemort)
(Voldemort/
Redis)
GTI
(Voldemort)
MRI
(Redis)

Architecture: Pull

Originator Index (ORI)

NO Push to MRIs at all

➡ 1 Message + 1 Originator Index Entry

Special profiles and some users have >500 friends

➡ get >500 ORIs on read => stress the system

Architecture: PushPull

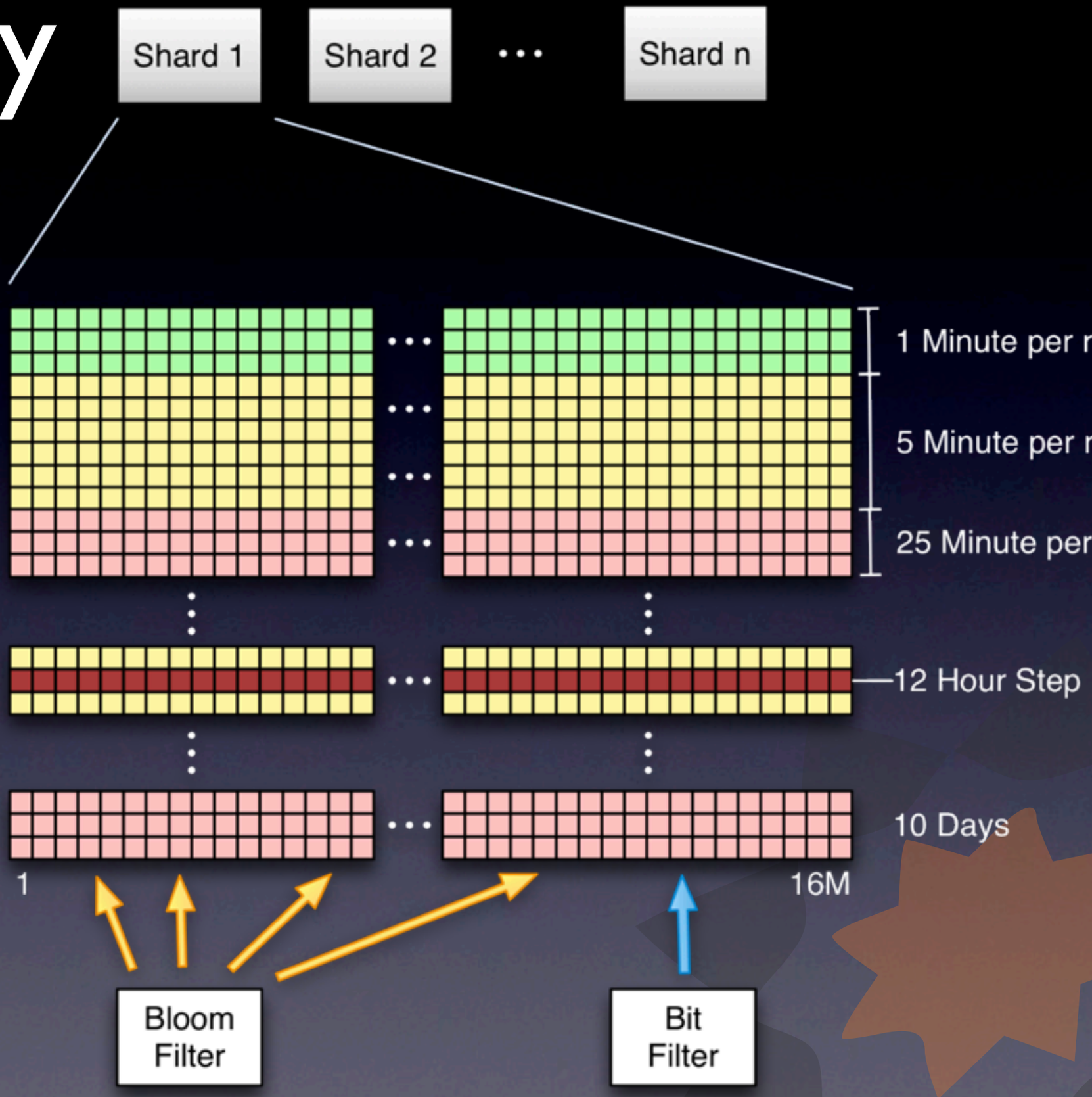
ORI + MRI

- Identify Users with recipient lists >{limit}
- Only push updates with recipients <{limit} to MRI
- Pull special profiles and users with >{limit} from ORI
- Identify active users with a bloom/bit filter for pull

Activity Filter

- Reduce read operations on storage
- Distinguish user activity levels
- In memory and shared across keys and types
- Scan full day of updates for16M users on a per minute granularity for 1000 friends in < 100msecs

Activity Filter



NoSQL


Redis
HBASE
mongoDB
{name: "mongo", type: "db"}
ORACLE
BERKELEY DB
riak
Neo4j
the graph database
Cassandra
apache
CouchDB
relax
HAZELCAST
SOFTWARE
Project Voldemort
A distributed database.

NoSQL: Redis

ORI + MRI on Steroids

- Fast in memory Data-Structure Server
- Easy protocol
- Asynchronous Persistence
- Master-Slave Replication
- Virtual-Memory
- JRedis - The Java client

NoSQL: Redis

ORI + MRI on Steroids

Data-Structure Server

- Datatypes: String, List, Sets, ZSets
- We use ZSets (sorted sets) for the Push Recipient Indexes

```
Insert

    for (recipient : recipients) {
         jredis.zadd(recipient.id, streamEntryIndex);
     }

Get

     jredis.zrange(streamOwnerId, from, to)
     jredis.zrangebyscore(streamOwnerId, someScoreBegin, someScoreEnd)
```

NoSQL: Redis

ORI + MRI on Steroids

Persistence - AOF and Bgsave

AOF - append only file
- append on operation

Bgsave - asynchronous snapshot
- configurable (timeperiod or every n operations)
- triggered directly

We use AOF as it‘s less memory hungry
Combined with bgsave for additional backups

NoSQL: Redis

ORI + MRI on Steroids

Virtual - Memory

Storing Recipient Indexes for 16 mio users à ~500 entries would lead to >250 GB of RAM needed

With Virtual Memory activated Redis swaps less frequented values to disk

➡ Only your hot dataset is in memory

➡ 40% logins per day / only 20% of these in peak
~ 20GB needed for hot dataset

NoSQL: Redis

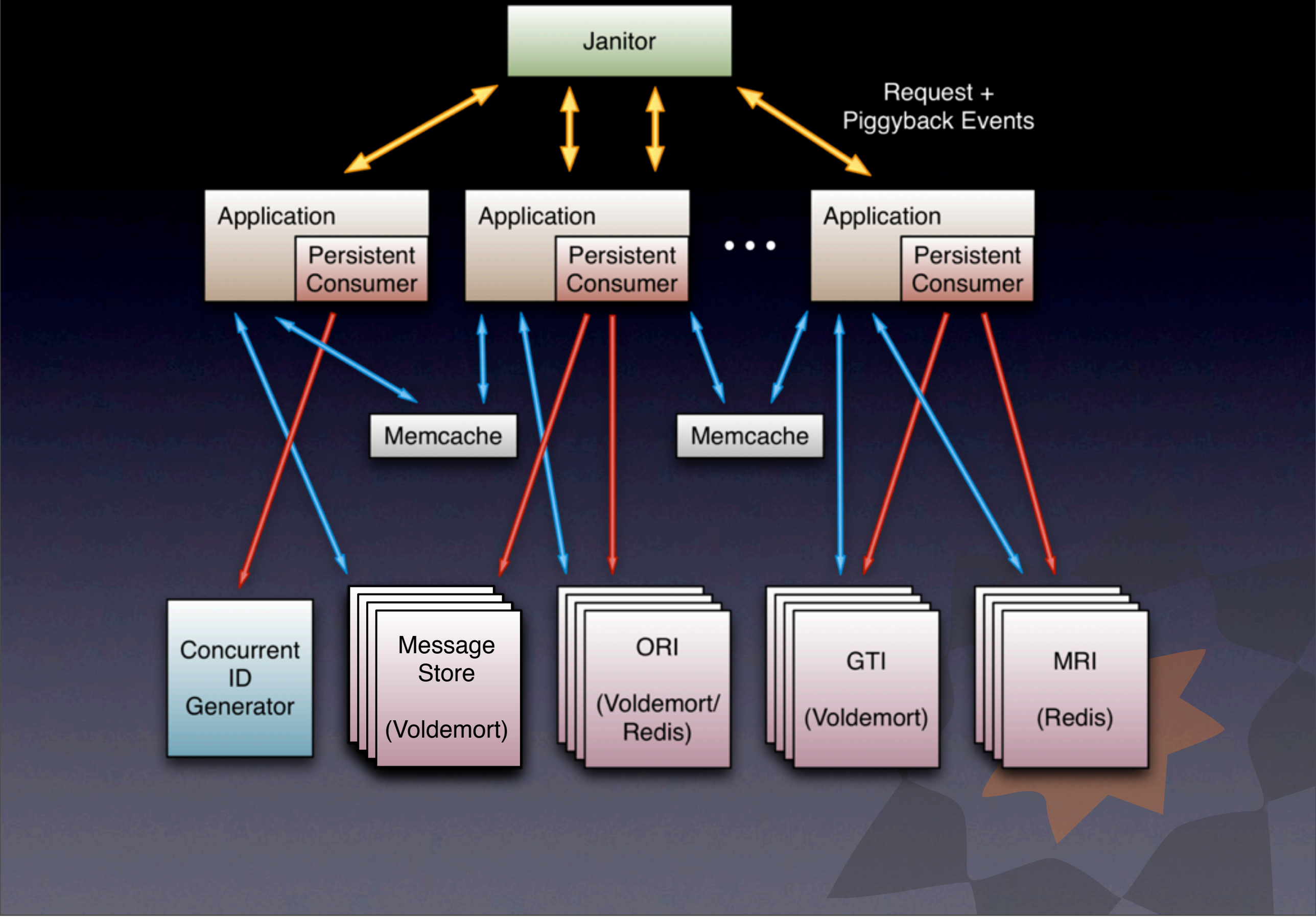
ORI + MRI on Steroids

Jredis - Redis java client

- Pipelining support (sync and async semantics)
- Redis 1.2.3 compliant

The missing parts

- No consistent hashing
- No rebalancing


Janitor
Request +
Piggyback Events
Application
Persistent
Consumer
Application
Persistent
Consumer
Application
Persistent
Consumer
Memcache
Memcache
Concurrent
ID
Generator
Message
Store
(Voldemort)
ORI
(Voldemort/
Redis)
GTI
(Voldemort)
MRI
(Redis)

Janitor

Request +
Piggyback Events

Application
Persistent Consumer

Application
Persistent Consumer

Application
Persistent Consumer

Memcach

Message Store

(Voldemort)

Concurrent ID Generator

Message Store

(Voldemort)

(Voldemort/ Redis)

GTI

(Voldemort)

MRI

(Redis)

NoSQL: Voldemort

No #fail Messagestore (MS)

- Key-Value Store
- Replication
- Versioning
- Eventual Consistency
- Pluggable Routing / Hashing Strategy
- Rebalancing
- Pluggable Storage-Engine

NoSQL: Voldemort

No #fail Messagestore (MS)

Configuring replication, reads and writes

```
<store>
    <name>stream-ms</name>
    <persistence>bdb</persistence>
    <routing>client</routing>
    <replication-factor>3</replication-factor>
    <required-reads>2</required-reads>
    <required-writes>2</required-writes>
    <prefered-reads>3</prefered-reads>
    <prefered-writes>3</prefered-writes>
    <key-serializer><type>string</type></key-serializer>
    <value-serializer><type>string</type></value-serializer>
    <retention-days>8</retention-days>
</store>
```

NoSQL: Voldemort

No #fail Messagestore (MS)

Write:

```
client.put(key, myVersionedValue);
```

Update(read-modify-write):

```
public class MriUpdateAction extends UpdateAction<String, String> {

    public MriUpdateAction(String key, ItemIndex index) {
        this.key = key;
        this.index = index;
    }

    @Override
    public void update(StoreClient<String, String> client) {
        Versioned<String> versionedJson = client.get(this.key);
        versionedJson.setObject("my value");
        client.put(this.key, versionedJson);
    }
}
```

NoSQL: Voldemort

No #fail Messagestore (MS)

Eventual Consistency - Read

```
public MriInconsistencyResolver implements InconsistencyResolver<Versioned<String>> {

    public List<Versioned<String>> resolveConflicts(List<Versioned<String>> items){
        Versioned<String> vers0 = items.get(0);
        Versioned<String> vers1 = items.get(1);
        if(vers0 == null && vers1 == null) {
            return null;
        }

        List<Versioned<String>> li = new ArrayList<Versioned<String>>(1);
        if(vers0 == null) {
            li.add(vers1);
            return li;
        }
        if(vers1 == null) {
            li.add(vers0);
            return li;
        }

        // resolve your inconsistency here e.g. merge to lists
    }
}
```

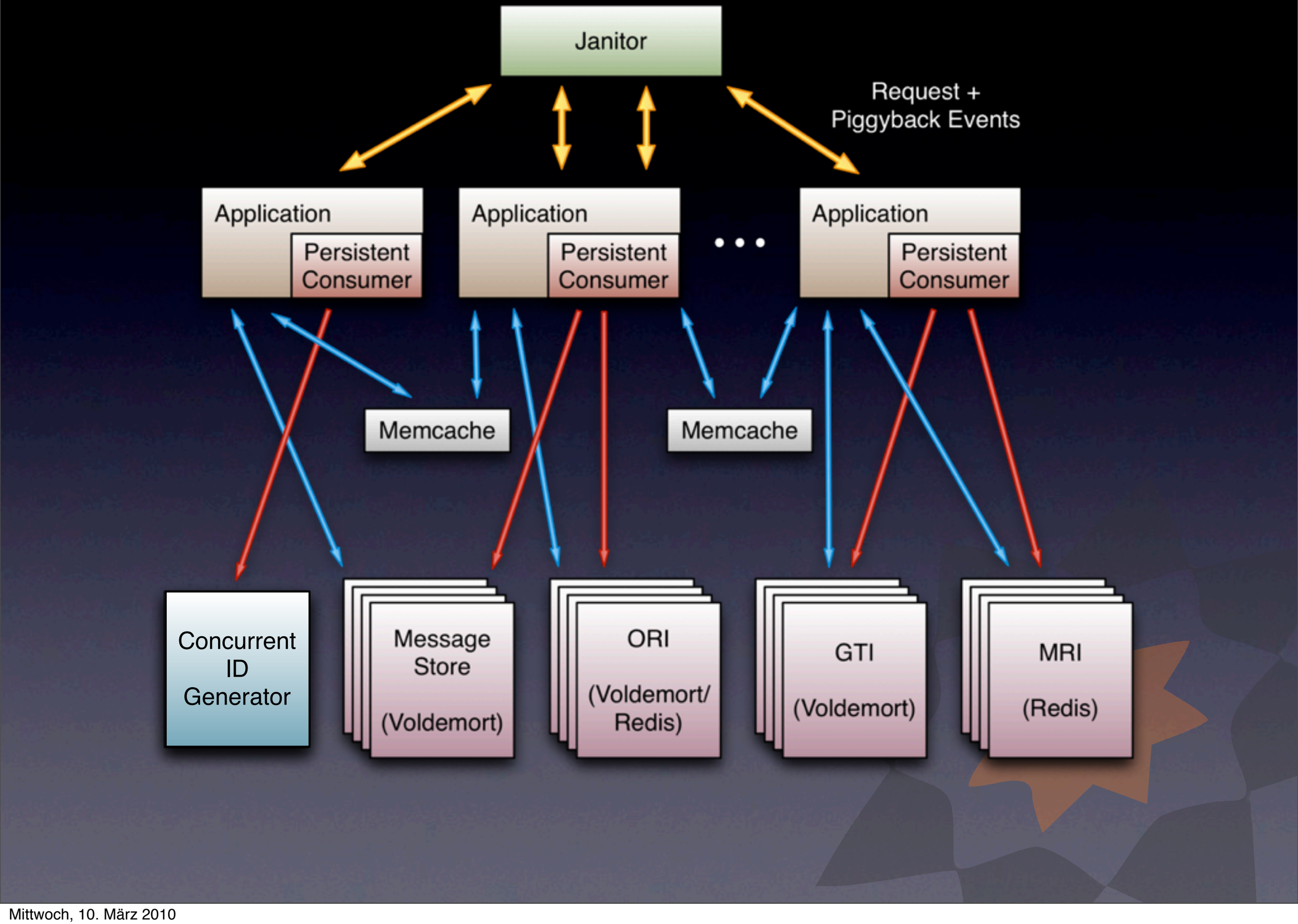
The default inconsistency resolver automatically takes the newer version

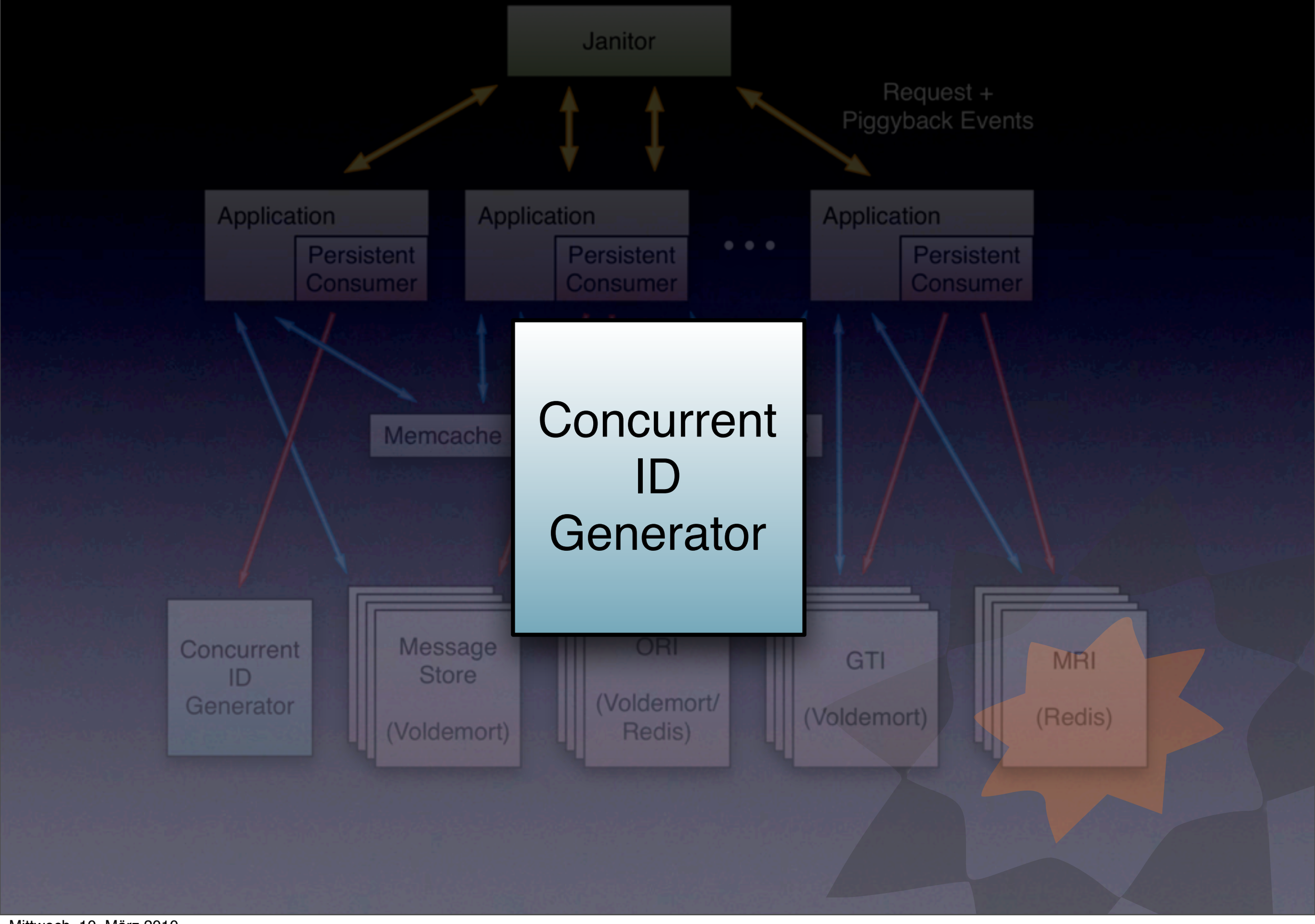
NoSQL:Voldemort

No #fail Messagestore (MS)

Configuration

- Choose a big number of partitions
- Reduce the size of the BDB append log
- Balance Client and Server Threadpools


Janitor
Request +
Piggyback Events
Application
Persistent
Consumer
Application
Persistent
Consumer
Application
Persistent
Consumer
Memcache
Memcache
Concurrent
ID
Generator
Message
Store
(Voldemort)
ORI
(Voldemort/
Redis)
GTI
(Voldemort)
MRI
(Redis)


Janitor
Request +
Piggyback Events
Application
Persistent
Consumer
Application
Persistent
Consumer
Application
Persistent
Consumer
Memcache
Concurrent
ID
Generator
Concurrent
ID
Generator
Message
Store
(Voldemort)
ORI
(Voldemort/
Redis)
GTI
(Voldemort)
MRI
(Redis)

NoSQL: Hazelcast

Concurrent ID Generator (CIG)

- In Memory Data Grid
- Dynamically Scales
- Distributed java.util.{Queue|Set|List|Map}
  and more
- Dynamic Partitioning with Backups
- Configurable Eviction
- Persistence

NoSQL: Hazelcast

Concurrent ID Generator (CIG)

Cluster-wide ID Generation:

- No UUID because of architecture constraint
- IDs of Stream Entries generated via Hazelcast
- Replication to avoid loss of count
- Background persistence used for disaster recovery

NoSQL: Hazelcast

Concurrent ID Generator (CIG)

Generate Unique Sequencial Numbers (Distributed Autoincrement):

- Nodes get ranges assigned (node1: 10000-19999, node2: 20000-29999 ID's)
- IDs per range locally incremented on the node (thread safe/atomic)
- Distributed locks secure range assignment for nodes

NoSQL: Hazelcast

Concurrent ID Generator (CIG)

Example Configuration

```
<map name=".vz.stream.CigHazelcast">
        <map-store enabled="true">

          <class-name>net.vz.storage.CigPersister</class-name>
          <write-delay-seconds>0</write-delay-seconds>

        </map-store>

        <backup-count>3</backup-count>

</map>
```

NoSQL: Hazelcast

Concurrent ID Generator (CIG)

Future use-cases:

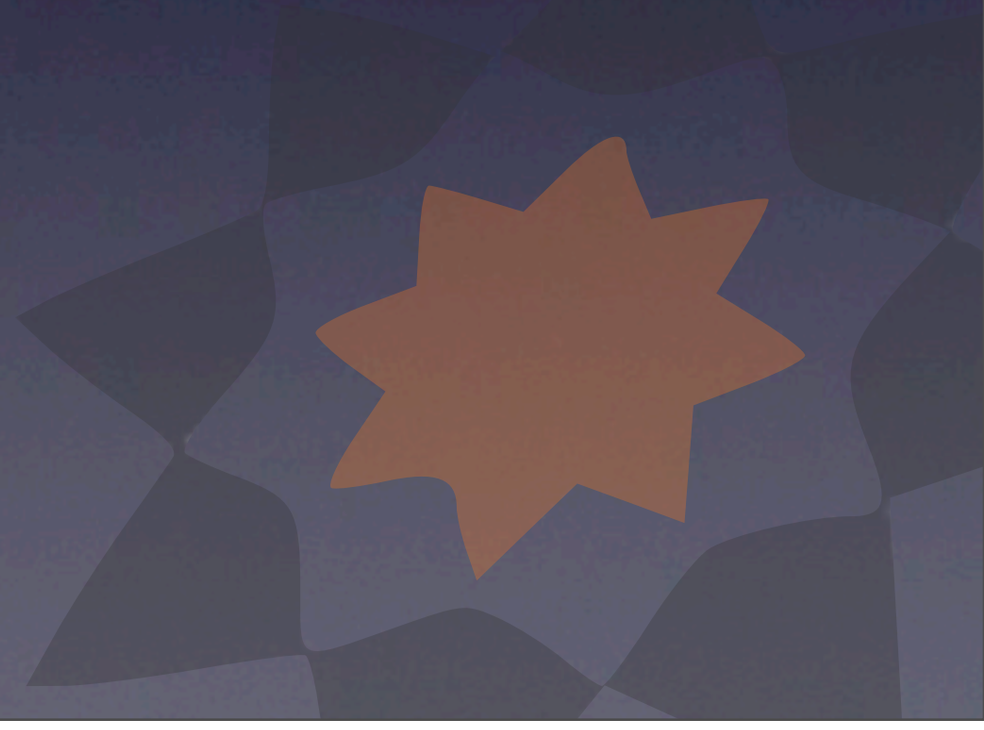
- Advanced preaggregated cache
- Distributed Executions

Lessons Learned

- Start benchmarking and profiling your app early!
- A fast and easy Deployment keeps the motivation up
- Configure Voldemort carefully (especially on large heap machines)
- Read the mailing lists of the #nosql system you use
- No Solution in docs? - read the sources!
- At some point stop discussing and just do it!

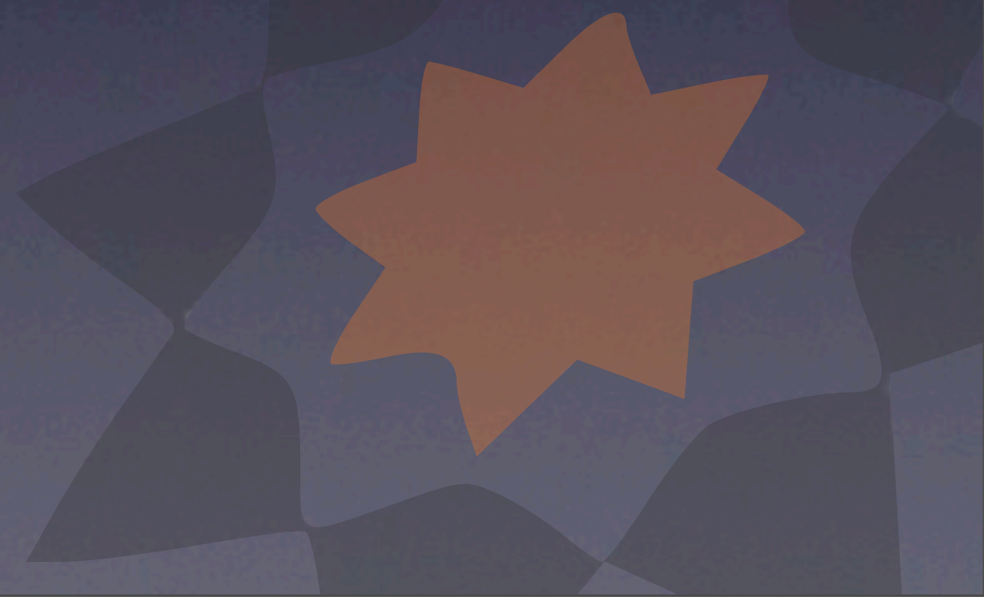
In Progress

- Network Stream
- Global Public Stream
- Stream per Location
- Hashtags



Future

- Geo Location Stream
- Third Party API



Questions?