King

Every day is like a little DDOS attack

A billion games played per day

Lars Sjödin, Server Architect, King

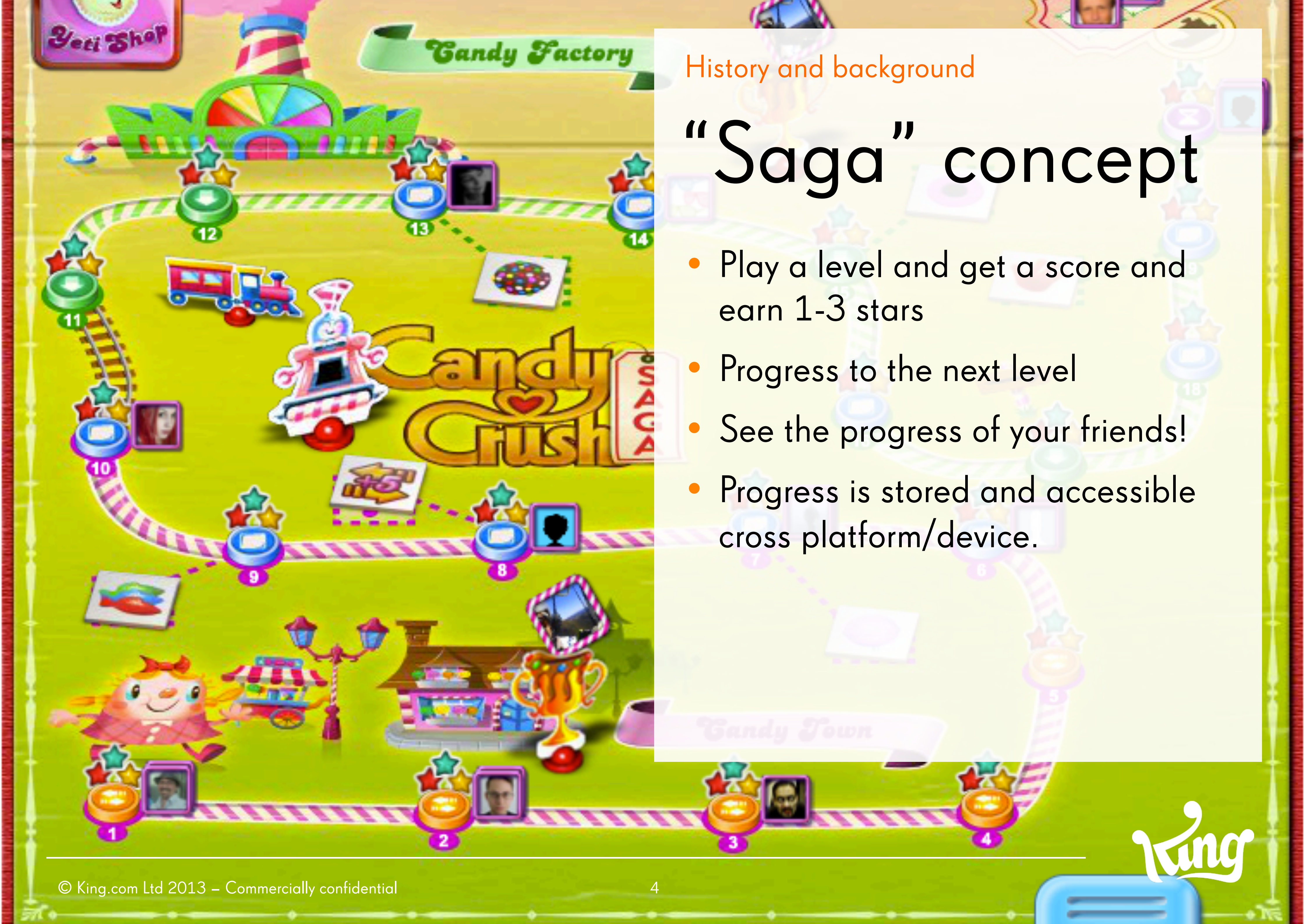
History and background

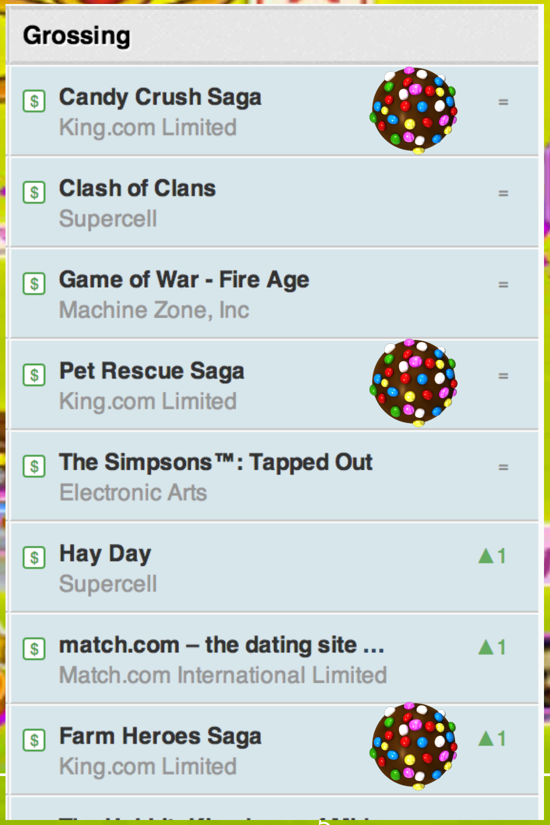
”Interactive entertainment company”

- Create own brands and game concepts
- Develop, design and publish the actual game
- Market and sell directly to players
- Cooperate with social graph providers (mainly Facebook) and distributors (Apple, Google, Amazon) to reach and engange players

"Saga" concept

- Play a level and get a score and earn 1-3 stars
- Progress to the next level
- See the progress of your friends!
- Progress is stored and accessible cross platform/device.





Grossing

Candy Crush Saga King.com Limited		=
Clash of Clans Supercell		=
Game of War - Fire Age Machine Zone, Inc		=
Pet Rescue Saga King.com Limited		=
The Simpsons™: Tapped Out Electronic Arts		=
Hay Day Supercell		▲1
match.com – the dating site ... Match.com International Limited		▲1
Farm Heroes Saga King.com Limited		▲1

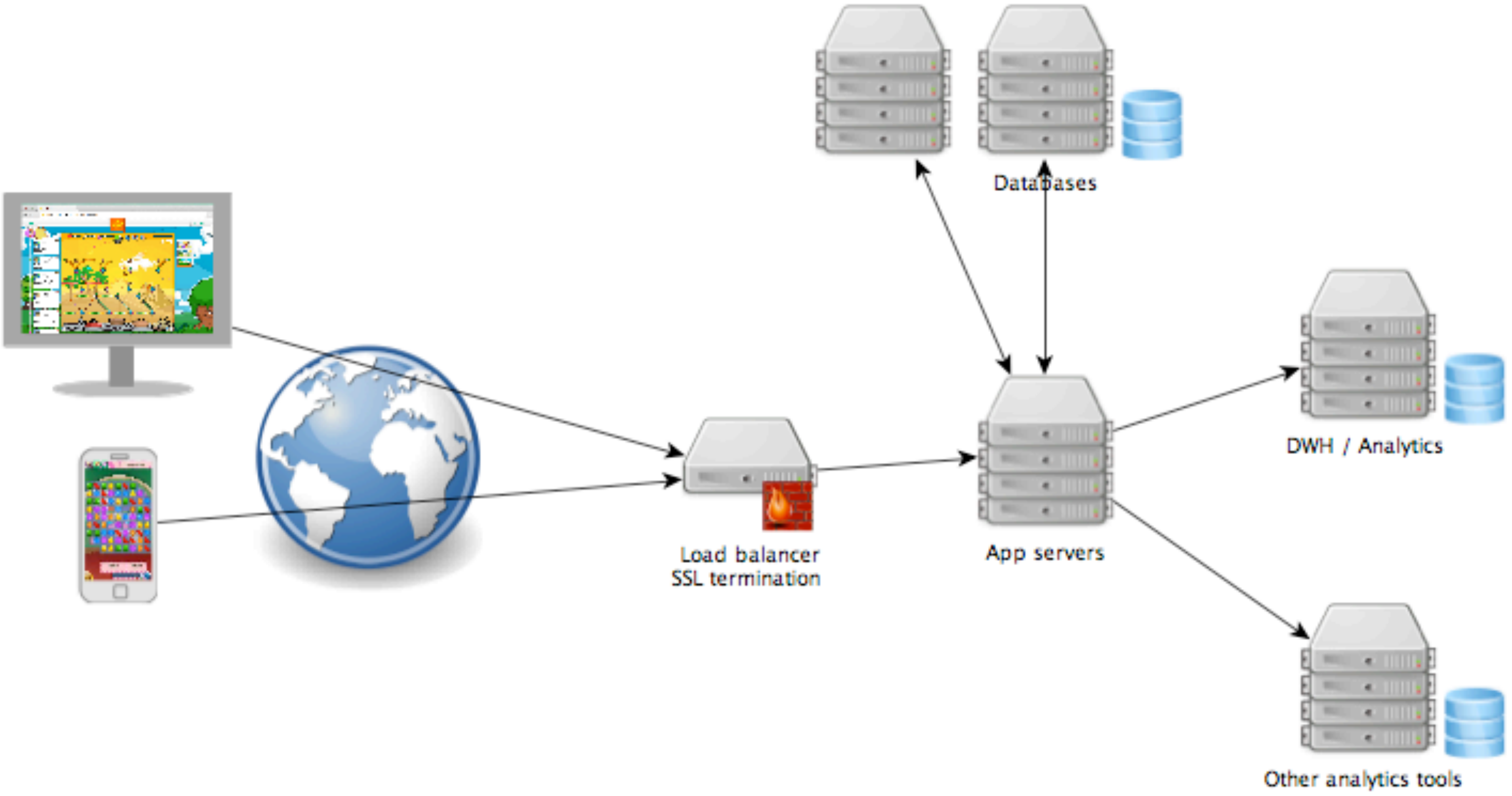
Emergency Alert
Flash Flood Warning this area til 6:00 PM CDT. Avoid flood areas. Check local media. -NWS

History and background

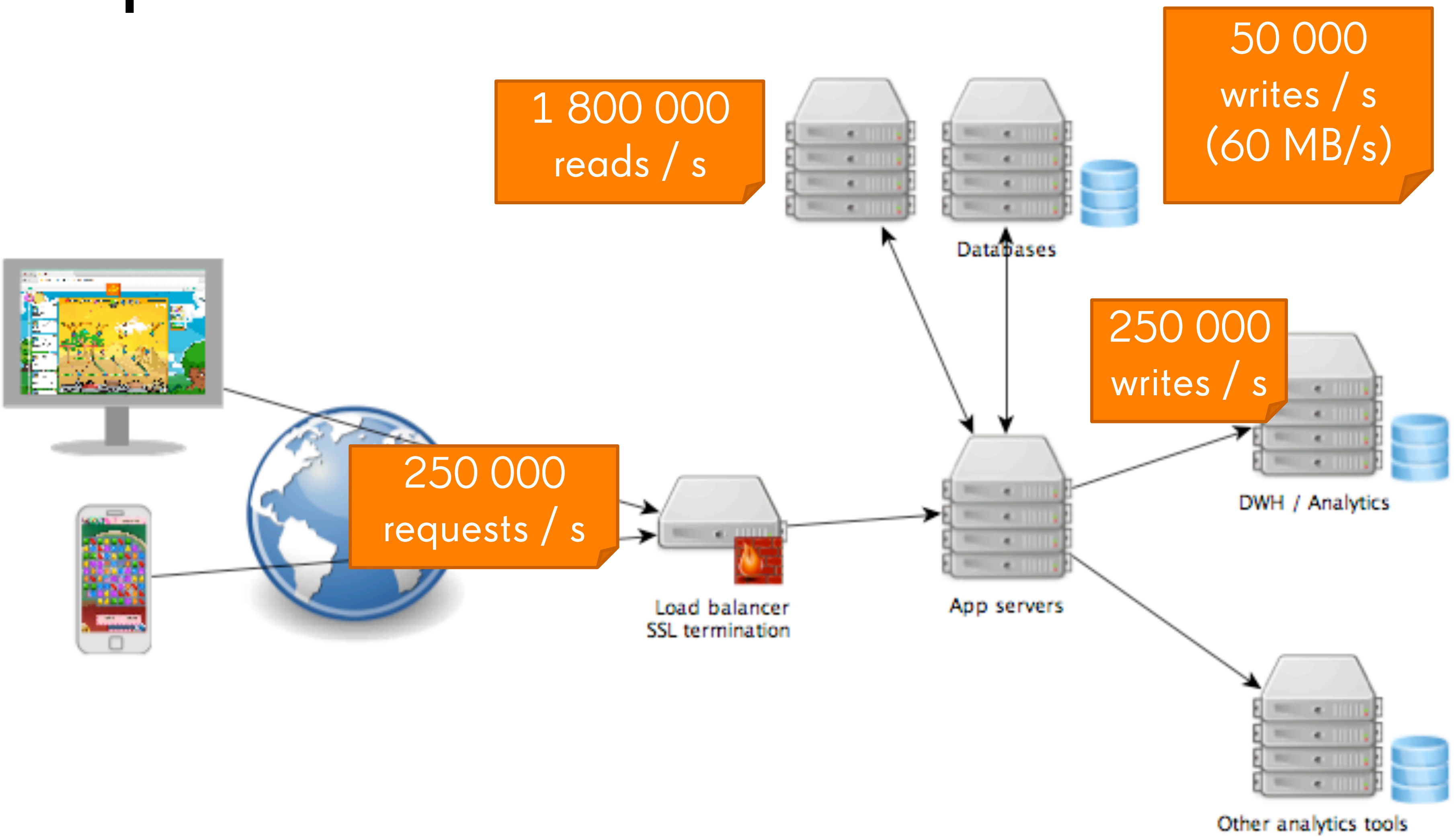
Since 2003

- August 2003 - Launch of the first gaming site called Midasplayer.com
- January 2007 - Over 80 million games played / month
- January 2009 - Over 350 million games played / month
- June 2013 - Over 1 billion games played / day

Setup and volumes



Setup and volumes



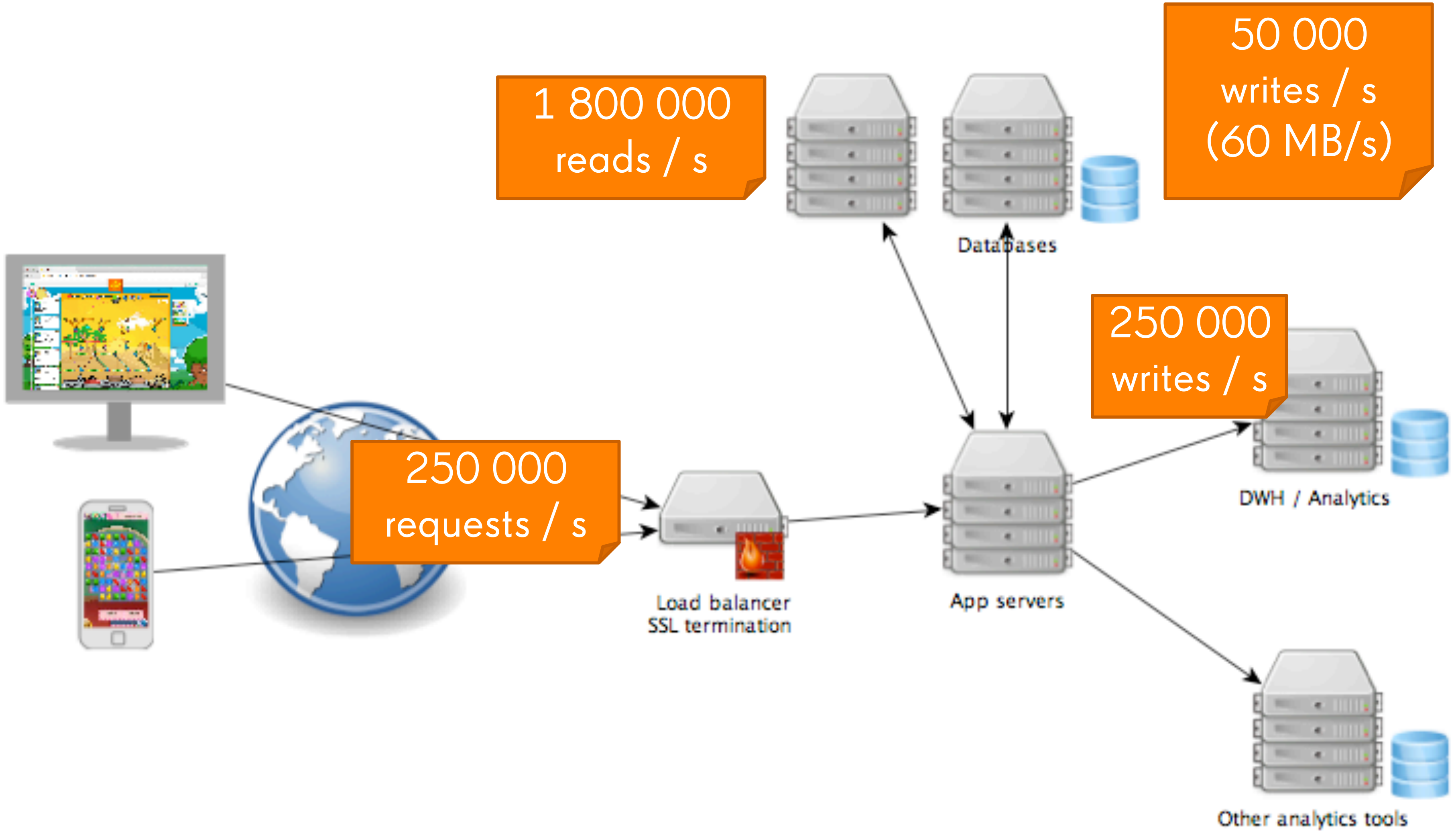
1 800 000 reads / s

50 000 writes / s (60 MB/s)

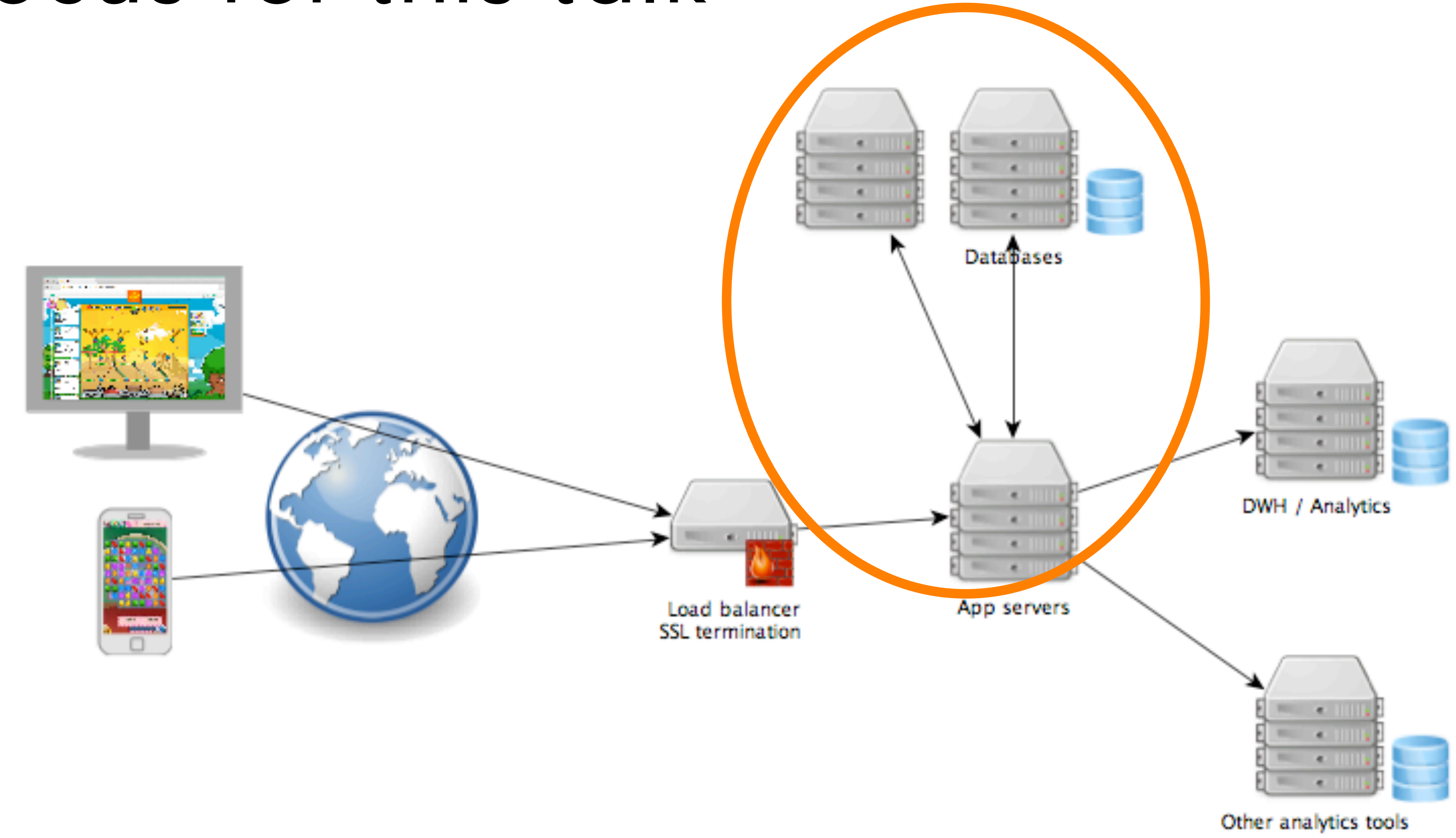
250 000 writes / s

250 000 requests / s

What to do?


1 800 000 reads / s
50 000 writes / s (60 MB/s)
Databases
250 000 writes / s
DWH / Analytics
250 000 requests / s
Load balancer SSL termination
App servers
Other analytics tools

Focus for this talk



First

Eliminate unrealistically strict requirements

Cut yourself some slack:

- If each player plays 10 games a day, what is realistically the impact of failing to store 1 out of 100 000 000? (ie, stop the redundancy and availability hysteria somewhere)

First

Eliminate unrealistically strict requirements

Cut yourself some slack:

- If each player plays 10 games a day, what is realistically the impact of failing to store 1 out of 100 000 000? (ie, stop the redundancy and availability hysteria somewhere)
- The player only needs to see fresh versions of it's own data. (Unless you design a game that forces players to sit next to each other or require immediate response)

First

Eliminate unrealistically strict requirements

Cut yourself some slack:

- If each player plays 10 games a day, what is realistically the impact of failing to store 1 out of 100 000 000? (ie, stop the redundancy and availability hysteria somewhere)
- The player only needs to see fresh versions of it's own data. (Unless you design a game that forces players to sit next to each other or require immediate response)
- If you sell virtual goods, don't use database transactions, prefer storing things in the order most beneficial for the player.

First

Eliminate unrealistically strict requirements

Cut yourself some slack:

- If each player plays 10 games a day, what is realistically the impact of failing to store 1 out of 100 000 000? (ie, stop the redundancy and availability hysteria somewhere)
- The player only needs to see fresh versions of it's own data. (Unless you design a game that forces players to sit next to each other or require immediate response)
- If you sell virtual goods, don't use database transactions, prefer storing things in the order most beneficial for the player.
- Be reasonable: even the cash register of the local supermarket won't add up 100% by the end of the day, and it's still OK!

First

Eliminate unrealistically strict requirements

Cut yourself some slack:

- If each player plays 10 games a day, what is realistically the impact of failing to store 1 out of 100 000 000? (ie, stop the redundancy and availability hysteria somewhere)
- The player only needs to see fresh versions of it's own data. (Unless you design a game that forces players to sit next to each other or require immediate response)
- If you sell virtual goods, don't use database transactions, prefer storing things in the order most beneficial for the player.
- Be reasonable: even the cash register of the local supermarket won't add up 100% by the end of the day, and it's still OK!

Conservative start

- Stuck to what we knew
- and built on top of that!

Conservative start

- Plain java server without hibernate or J2EE stuff
- A lot of homegrown libraries:
  - Caching
  - Sharded data storage
  - Database pool
  - Serialization and deserialization primitives and conventions

A bit like a cloud API

Conservative start

The butter and cream of the storage world:

MySQL

+

Memcached

Makes everything better!

DataStore

- Stores BLOBS in innoDB (mySQL)
- Key is a String (usually the UserID)
- Data is just Strings (bytes would have been better..)!

- String getData(DatabaseSession dbSession, String kingApp, String table, String key, boolean locked);
- void setData(DatabaseSession dbSession, String kingApp, String table, String key, String data);
- void delete(DatabaseSession dbSession, String kingApp, String table, String key);

JsonStore

- Stores JSON data in a DataStore.

- <T> T get(DatabaseSession dbSession, KingApp kingApp, String table, String key, Class<T> clazz);
- <T> void set(DatabaseSession dbSession, KingApp kingApp, String table, String key, T t);
- <T> void delete(DatabaseSession dbSession, KingApp kingApp, String table, String key);
- <T> T update(DatabaseSession dbSession, KingApp kingApp, String table, String key, Operation<T> operation);

```
public interface Operation<T> {
    T operate(T t);
}
```

UserJsonStore

- Stores stores data in a JsonStore using sharding information.

- <T> T get(KingApp kingApp, String table, UserStoreKey key, Class<T> clazz);
- <T> void set(KingApp kingApp, String table, UserStoreKey key, T t);
- <T> T update(KingApp kingApp, String table, UserStoreKey key, Operation<T> operation);

```
public class UserStoreKey {
    private final long userId;
    private final String key;
}
```

Cut support to a minimun

- Only allow certain datatypes when reading/writing data
  - bool, int, double, String, [], Class with fields with valid types
  - Makes changing serialization / transport much easier
  - Makes support in many languages easier

Some reused principles

Data compatibility

The compatibility promise:

- Missing fields are 0, null or false when read
- Extra fields are ignored when read

➔Objects can be upgraded to a new schema when read!

Data compatibility in practice

```
public class SocialUser {
    private byte[] firstname=”Lars”;
    private long userId=1014427147;
    private long birthdayDateMillis=121759200000;
}
```

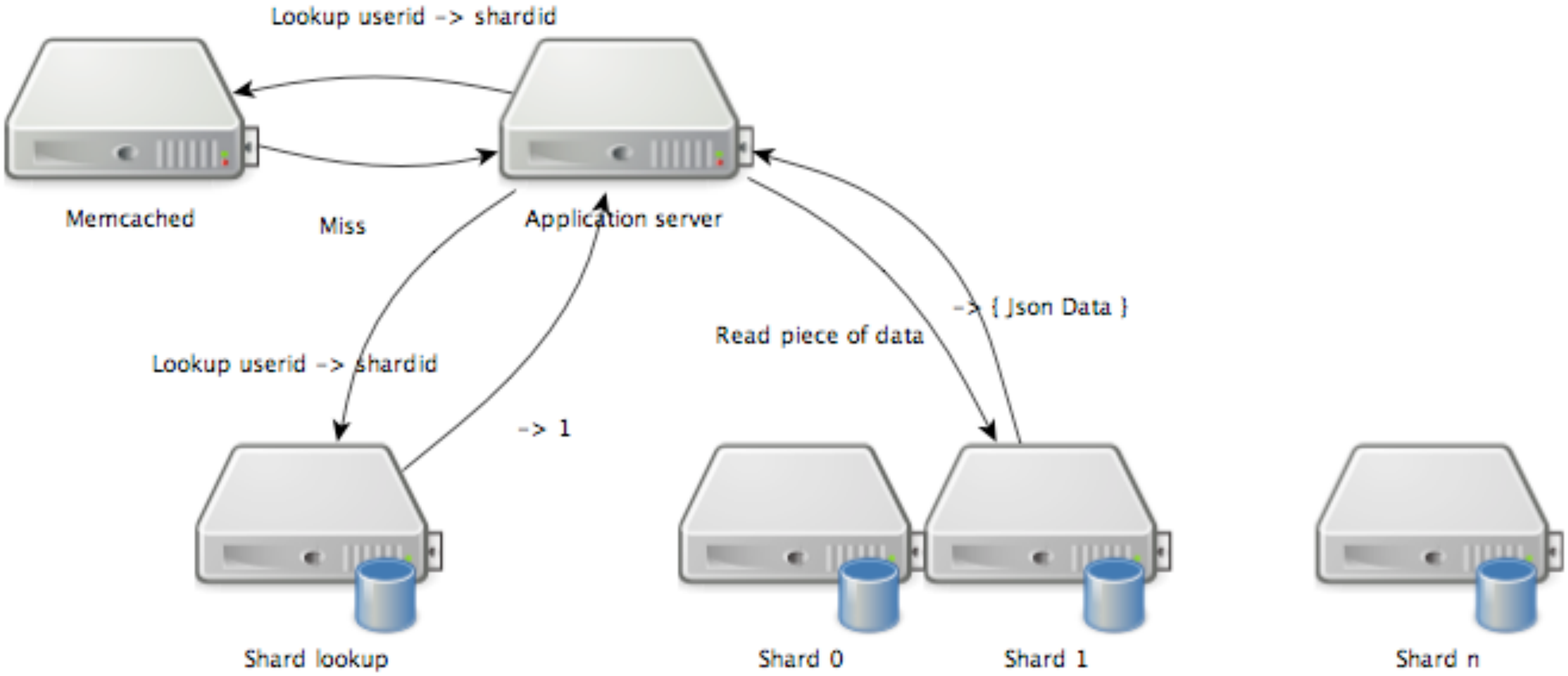
```
{
   ”firstname”:”Lars”,
   ”userId”:1014427147,
   “birthdayDateMillis”:121759200000
}
```

```
public class SocialUser {
    private byte[] firstname=”Lars”;
    private byte[] picSmall=null;
    private long userId=1014427147;
    private long birthdayDateMillis=121759200000;
}
```

Data compatibility

- Applied in the protocol which is JSON-RPC (http://json-rpc.org/)
- Applied when storing data in the DataStore
- Applied when storing objects in memcached

Regular operation



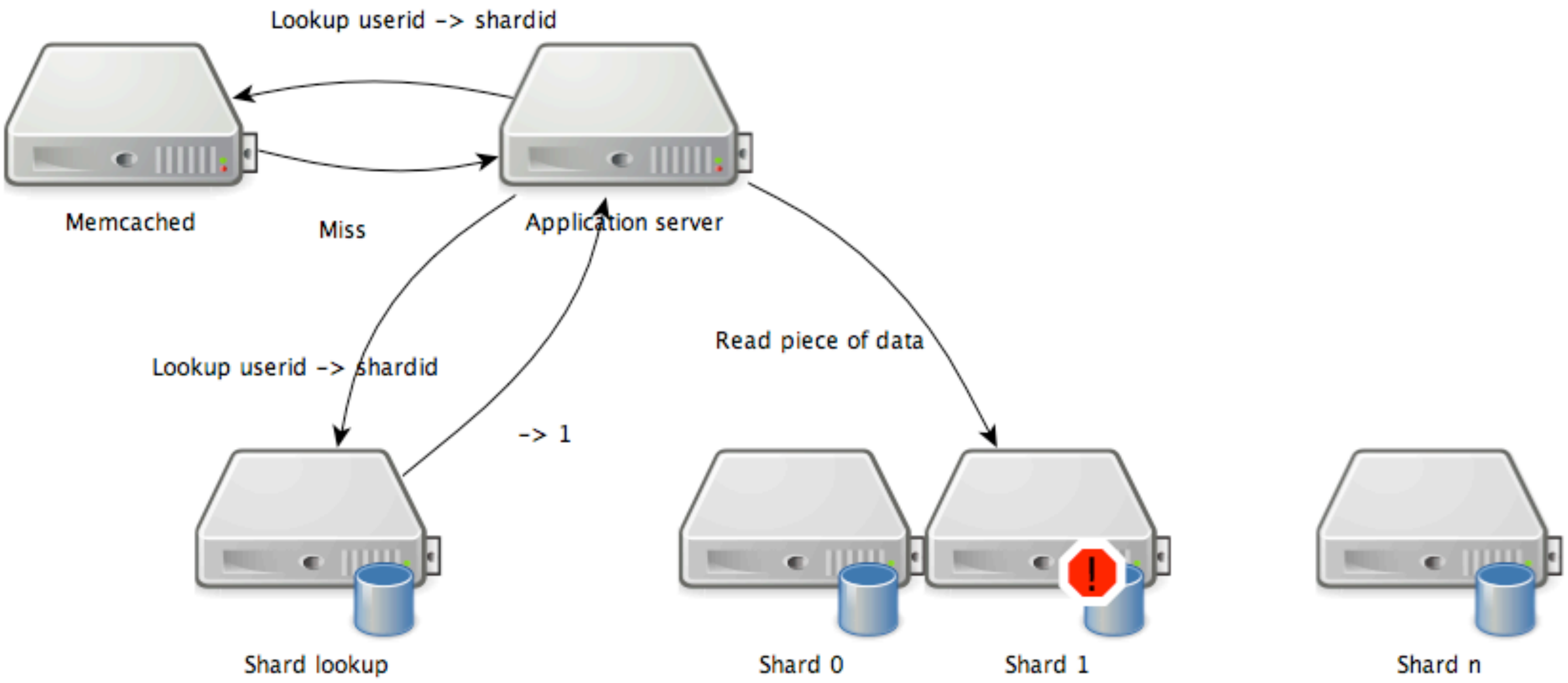
The connection pool

Many database connections needs management or an approach to prevent deadlocks:

1) Only use one connection and then return it immediately (But connection cycling is expensive)
2) Unbounded connection pools (really?)
3) "Connection order", ie always get connections to the databases in the same order (Works fine, needs enforcment)
4) Global "connection pool" (limits concurrency)

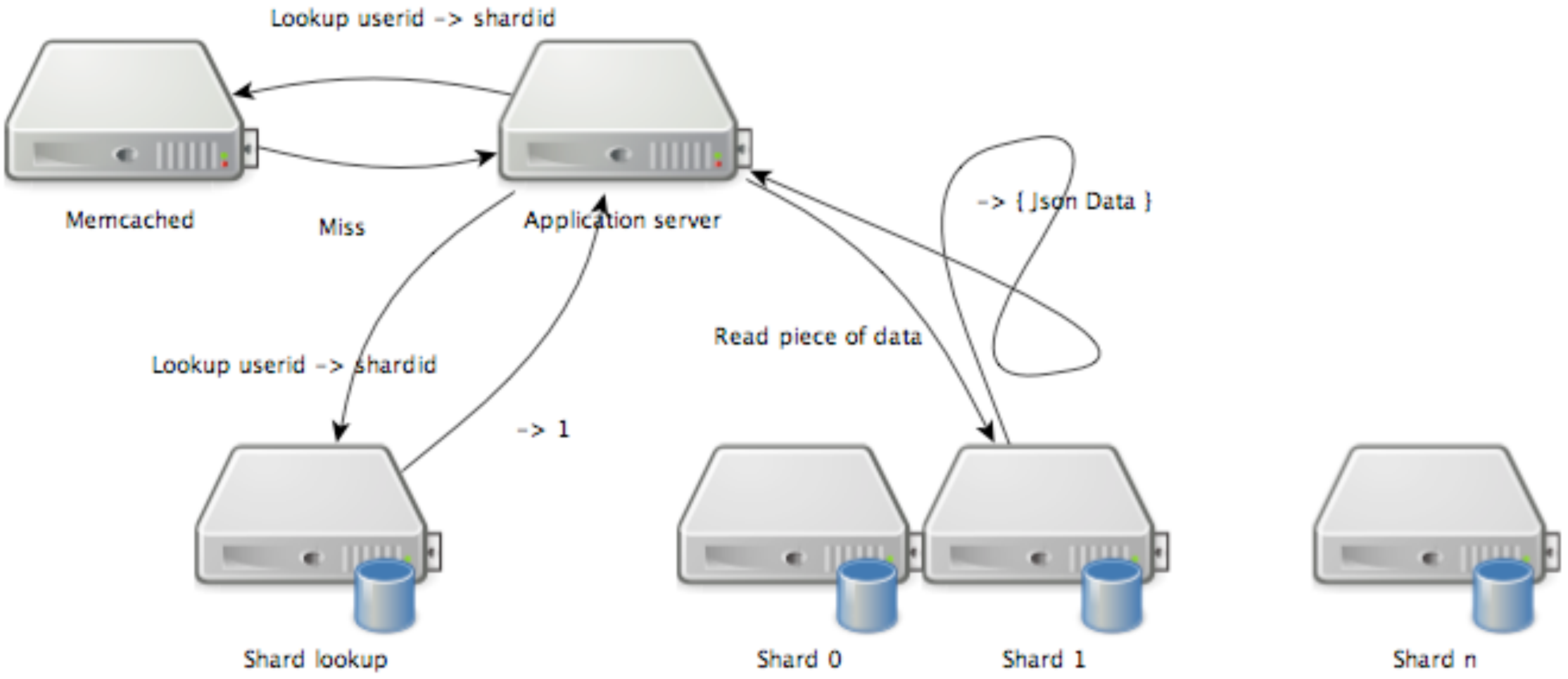
Problem scenarios

Plain failure of a shard. “Easy”, just leave out and retry “every now and then”



Problem scenarios

Slow shard. What to do?



Problem scenarios

Slow shard. What to do?

Monitor ALL queries as we go

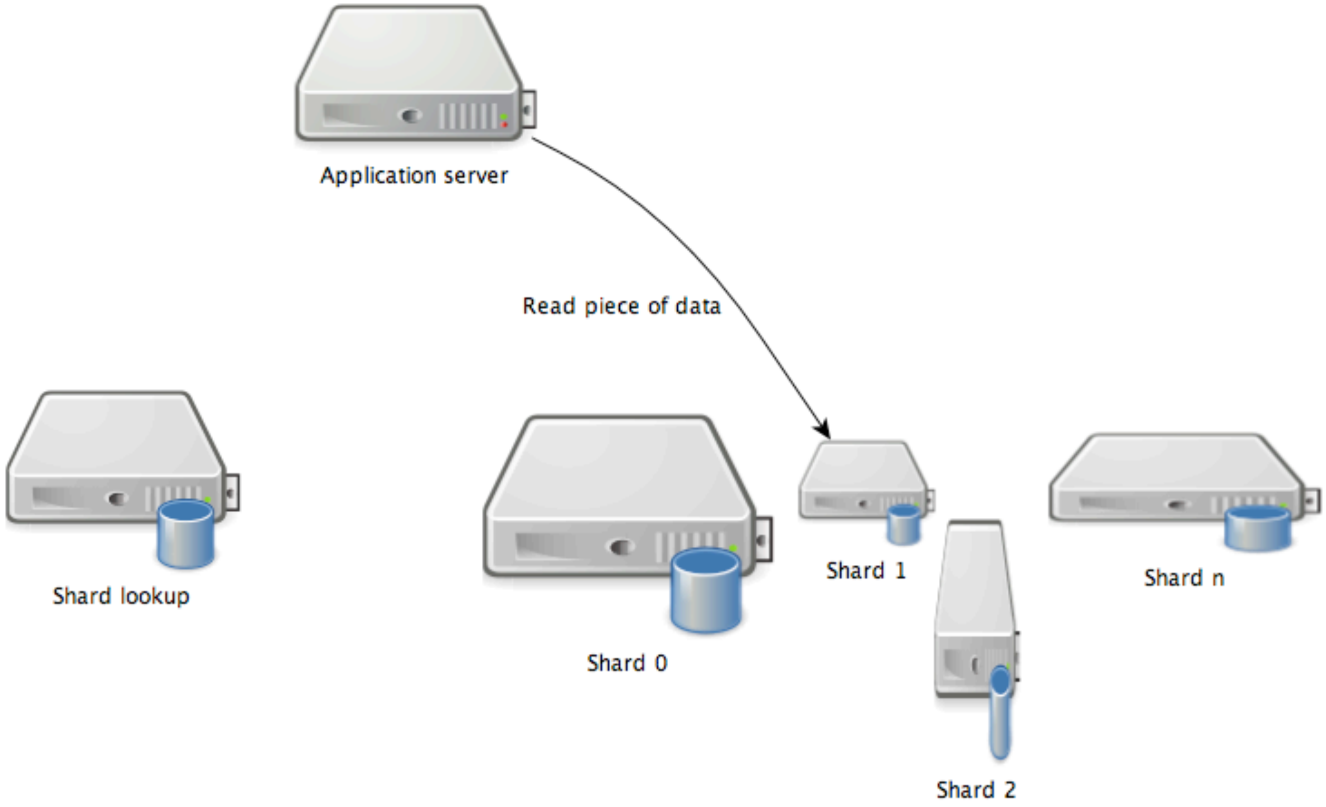
When queries start to be slow (more than 10 ms) we start measuring problems and throttle access to that shard.

Throttling = allow ONE connection to that shard to go through for monitoring purposes (but fail it for the one asking for the data...).

Measure query time and when it regains stable low values. Reopen the shard for business!

Problem scenarios

Heterogenous characteristics



Problem scenarios

Heterogenous characteristics

Reasons:

- Hardware is bought during different phases
- Old players are not as active as new players

Problem scenarios

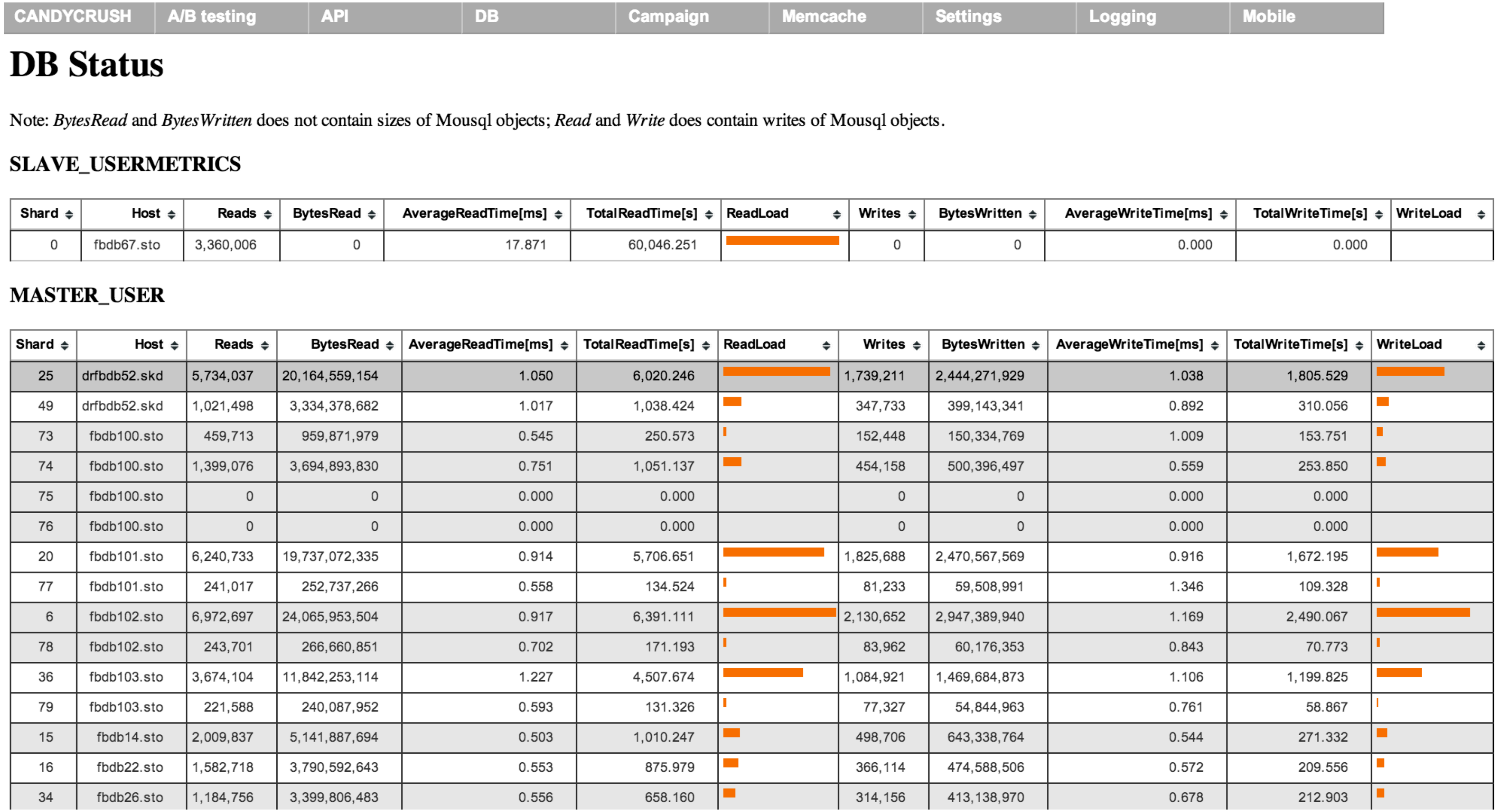
Heterogenous characteristics

Reasons:

- Hardware is bought during different phases
- Old players are not as active as new players
- A complex problem consisting of:
  Space left on disk
  Read and write performance
  Future plans

- Constant monitoring and rebalancing (Each server gets a scalar value based on query performance from Percona performance statistics)
- New players are manually configured to be created where we want them!

Monitoring of our databases!



DB Status

Note: *BytesRead* and *BytesWritten* does not contain sizes of Mousql objects; *Read* and *Write* does contain writes of Mousql objects.

SLAVE_USERMETRICS

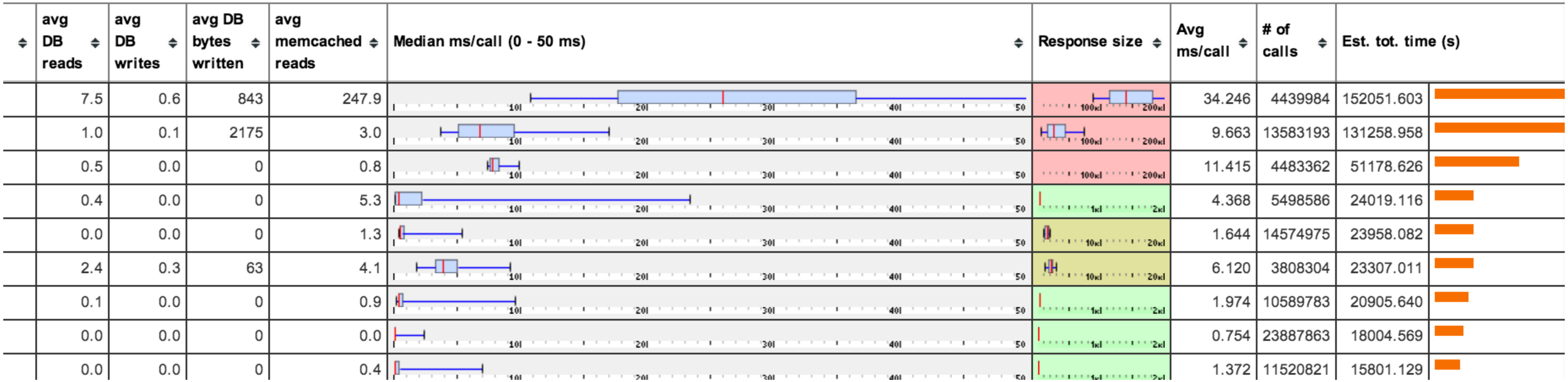
Shard	Host	Reads	BytesRead	AverageReadTime[ms]	TotalReadTime[s]	ReadLoad	Writes	BytesWritten	AverageWriteTime[ms]	TotalWriteTime[s]	WriteLoad
0	fbdb67.sto	3,360,006	0	17.871	60,046.251	▬	0	0	0.000	0.000	

MASTER_USER

Shard	Host	Reads	BytesRead	AverageReadTime[ms]	TotalReadTime[s]	ReadLoad	Writes	BytesWritten	AverageWriteTime[ms]	TotalWriteTime[s]	WriteLoad
25	drfbdb52.skd	5,734,037	20,164,559,154	1.050	6,020.246	▬	1,739,211	2,444,271,929	1.038	1,805.529	▬
49	drfbdb52.skd	1,021,498	3,334,378,682	1.017	1,038.424	▬	347,733	399,143,341	0.892	310.056	▬
73	fbdb100.sto	459,713	959,871,979	0.545	250.573	▬	152,448	150,334,769	1.009	153.751	▬
74	fbdb100.sto	1,399,076	3,694,893,830	0.751	1,051.137	▬	454,158	500,396,497	0.559	253.850	▬
75	fbdb100.sto	0	0	0.000	0.000		0	0	0.000	0.000	
76	fbdb100.sto	0	0	0.000	0.000		0	0	0.000	0.000	
20	fbdb101.sto	6,240,733	19,737,072,335	0.914	5,706.651	▬	1,825,688	2,470,567,569	0.916	1,672.195	▬
77	fbdb101.sto	241,017	252,737,266	0.558	134.524	▬	81,233	59,508,991	1.346	109.328	▬
6	fbdb102.sto	6,972,697	24,065,953,504	0.917	6,391.111	▬	2,130,652	2,947,389,940	1.169	2,490.067	▬
78	fbdb102.sto	243,701	266,660,851	0.702	171.193	▬	83,962	60,176,353	0.843	70.773	▬
36	fbdb103.sto	3,674,104	11,842,253,114	1.227	4,507.674	▬	1,084,921	1,469,684,873	1.106	1,199.825	▬
79	fbdb103.sto	221,588	240,087,952	0.593	131.326	▬	77,327	54,844,963	0.761	58.867	▬
15	fbdb14.sto	2,009,837	5,141,887,694	0.503	1,010.247	▬	498,706	643,338,764	0.544	271.332	▬
16	fbdb22.sto	1,582,718	3,790,592,643	0.553	875.979	▬	366,114	474,588,506	0.572	209.556	▬
34	fbdb26.sto	1,184,756	3,399,806,483	0.556	658.160	▬	314,156	413,138,970	0.678	212.903	▬

Monitoring of our system!

Monitoring resource usage for each request:



	avg DB reads	avg DB writes	avg DB bytes written	avg memcached reads	Median ms/call (0 - 50 ms)	Response size	Avg ms/call	# of calls	Est. tot. time (s)	
	7.5	0.6	843	247.9			34.246	4439984	152051.603	
	1.0	0.1	2175	3.0			9.663	13583193	131258.958	
	0.5	0.0	0	0.8			11.415	4483362	51178.626	
	0.4	0.0	0	5.3			4.368	5498586	24019.116	
	0.0	0.0	0	1.3			1.644	14574975	23958.082	
	2.4	0.3	63	4.1			6.120	3808304	23307.011	
	0.1	0.0	0	0.9			1.974	10589783	20905.640	
	0.0	0.0	0	0.0			0.754	23887863	18004.569	
	0.0	0.0	0	0.4			1.372	11520821	15801.129	

What happens with scale?

5-25 million daily active users (DAU)

What happens with scale?

5-25 million daily active users (DAU)

- Dealing with problems such as maintaining business during hardware failures.
  - A shard failure (or worse: slowdown), can damage overall system.
  - 3 (three!) connection pools have been tried. Own heuristics to kick a shard that is misbehaving

What happens with scale?

5-25 million daily active users (DAU)

- Adding hardware in a pace corresponding to growth
  - Hardware has real production lead and delivery times
  - Getting traffic estimates for 3 months ahead is hard
  - Order hardware for ”worst” (best?) case growth!
    - Ignore business estimates…

What happens with scale?

25+ million DAU

What happens with scale?

25+ million DAU

What happens with scale?

25+ million DAU

- Unboxing takes time and generates waste…
- Order racked hardware
- Optimize network infrastructure

What happens with scale?

25+ million DAU

- Unboxing takes time and generates waste...
- Order racked hardware
- Optimize network infrastructure
- Data "overflow" (always adding disk)
  - Backups
  - Event data

What happens with scale?

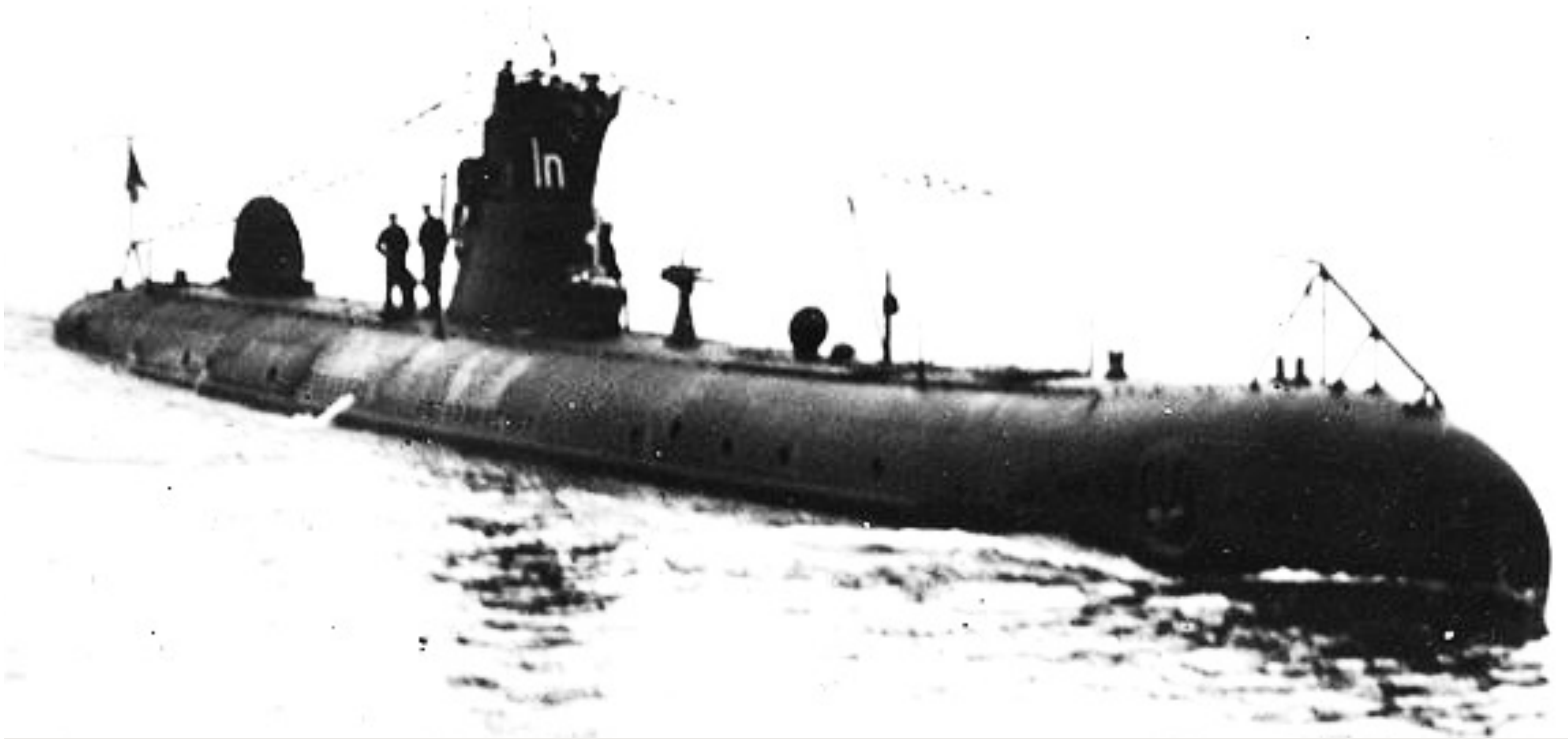
25+ million DAU

- Unboxing takes time and generates waste…
- Order racked hardware
- Optimize network infrastructure
- Data ”overflow” (always adding disk)
  - Backups
  - Event data

- Hardware generations! (ie: hetrogenous database cluster)
  - (Solution: background migrate users between shards based on performance heuristics!)

Launch cleverly

- Stakeholders on board when going live!
- "Test" live and measure while doing it!



Memcached

Every day is like a little DDOS attack

Thank you

That’s it

Questions?

(We’re hiring, check out http://about.king.com/)

lars.sjodin@king.com