Getting real with erlang

From the idea to a live system

Knut Nesheim @knutin

Paolo Negri @hungryblank

wooga

world of gaming

Social Games

Flash client (game)

HTTP API

http://www.flickr.com/photos/theplanetdotcom/4879421344

wooga
world of gaming

Social Games

HTTP API

- @ 1 000 000 daily users
- 5000 HTTP reqs/sec
- more than 90% writes

wooga
world of gaming

November 2010

- At the erlang user group!
- Learn where/how erlang is used
- 0 lines of erlang code across all @wooga code base

Why looking into erlang?

HTTP API

- @ 1 000 000 daily users
- 5000 HTTP reqs/sec
- around 60000 queries/sec

wooga
world of gaming

60000 qps

- Most maintenance effort in databases
- mix of SQL / NoSQL
- Already using in RAM data stores (REDIS)
- RAM DB are fast but expensive / high maintenance

Social games data

- User data self contained
- Strong hot/cold pattern - gaming session
- Heavy write load (caching is ineffective)

User session

1. start session
2. game actions
3. end session

User data

1. load all
2. read/update many times
3. data becomes cold

wooga
world of gaming

User session

1. start session
2. game actions
3. end session

Erlang process

1. start (load state)
2. responds to messages (use state)
3. stop (save state)

User session DB usage

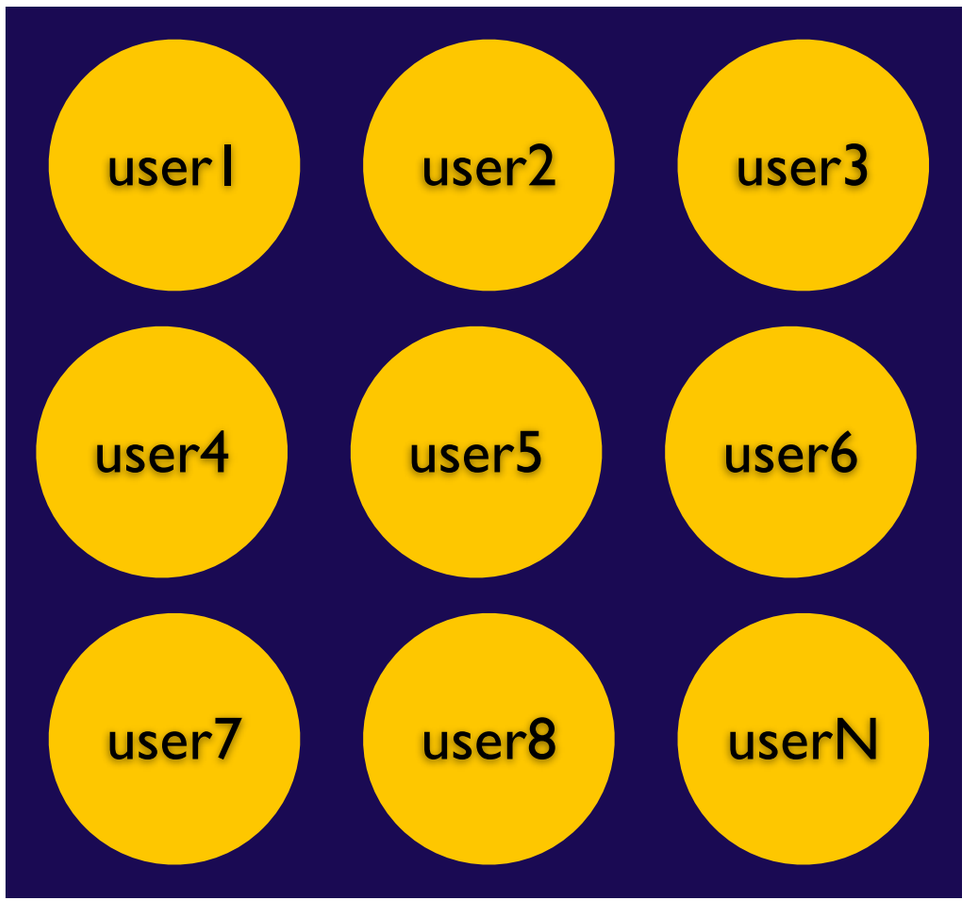
	Stateless server	Stateful server (Erlang)
start session	load user state	load user state
game actions	many queries	
end session		save user state

Erlang process

- Follows session lifecycle
- Contains and defends state (data)
- Acts as a lock/serializer (one message at a time)

December 2010

- Prototype



= user session

wooga
world of gaming

January 2011

- erlang team goes from 1 to 2 developers

Open topics

- Distribution/clustering
- Error handling
- Deployments
- Operations

Architecture

Architecture goals

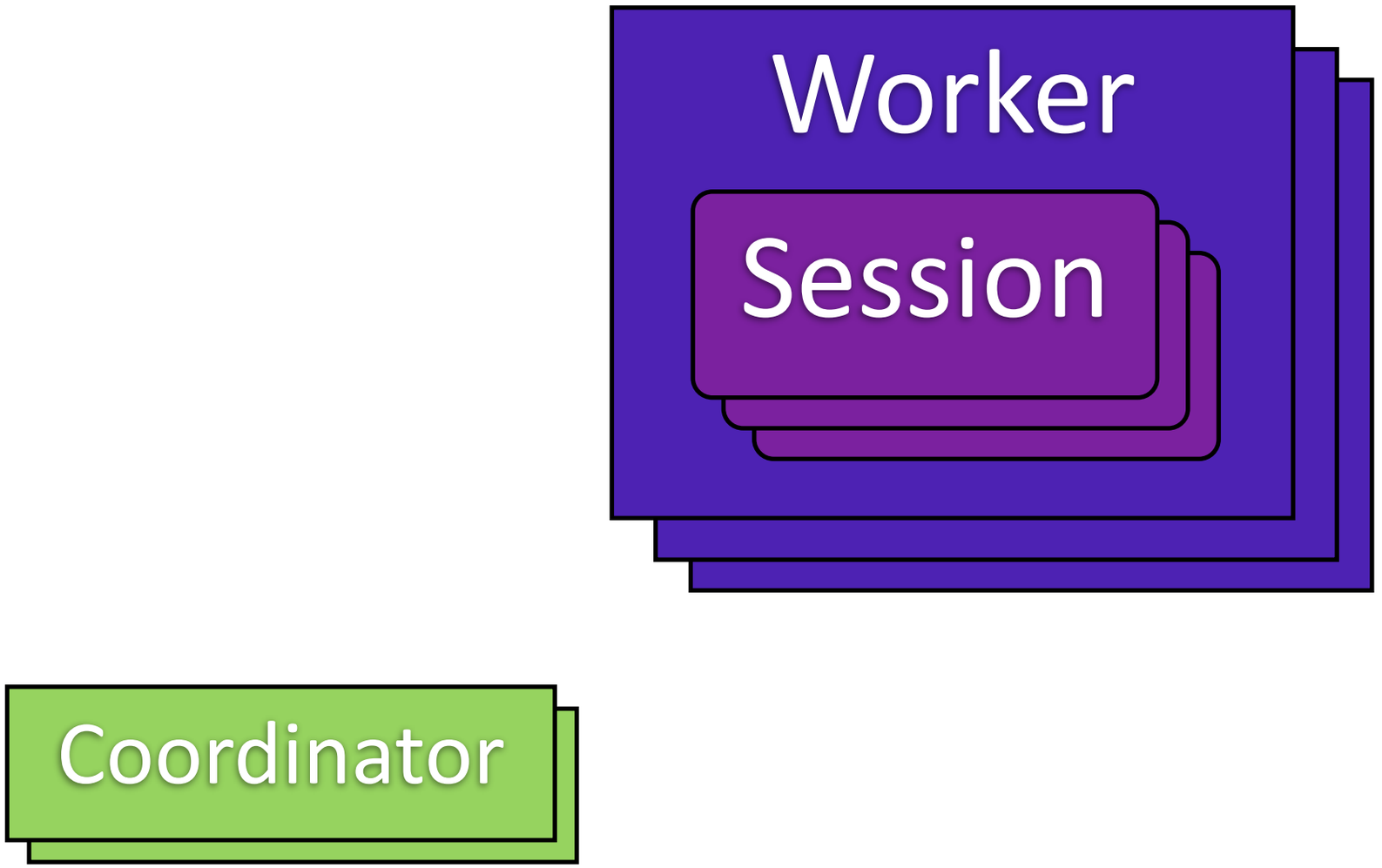
- Move data into the application server
- Be as simple as possible
- Graceful degradation when DBs go down
- Easy to inspect and repair state of cluster
- Easy to add more machines for scaling out

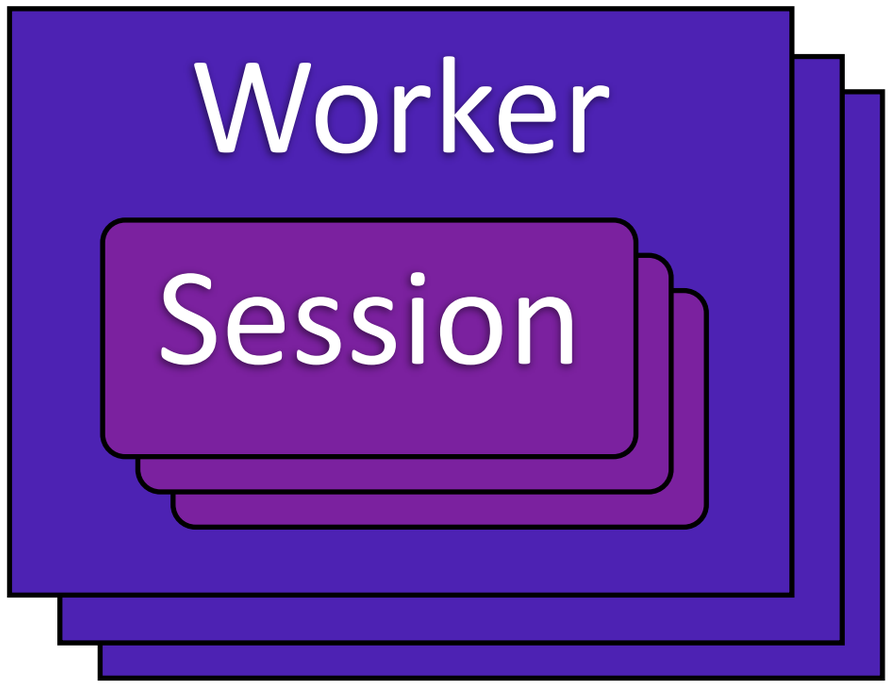
wooga
world of gaming

wooga
world of gaming


Worker
Session
Worker
Session
Worker
Session


wooga
world of gaming

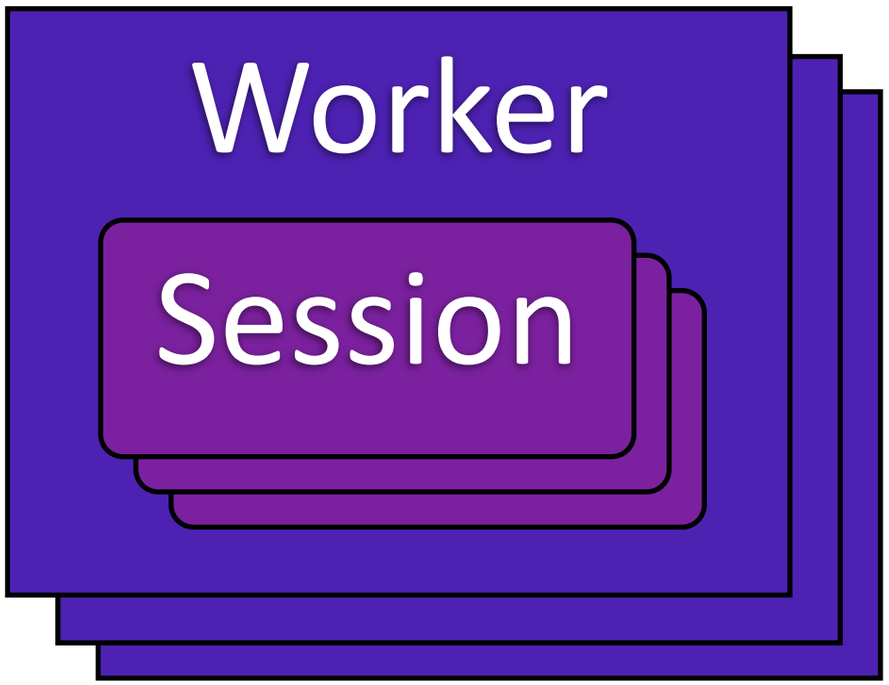

Worker
Session
Coordinator



Coordinator

Lock

redis



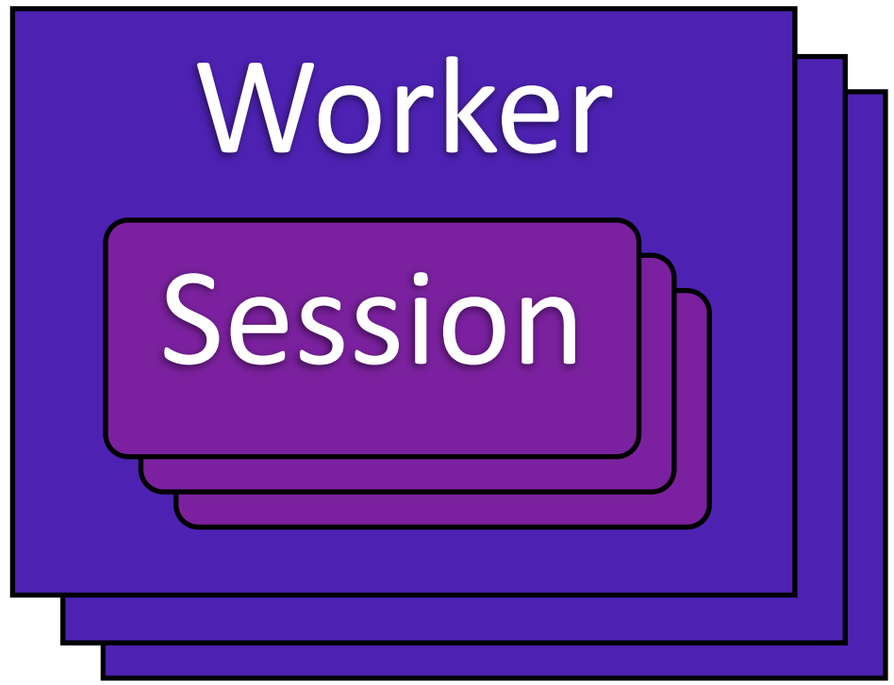
Coordinator

Lock

DBs

wooga
world of gaming

New user comes online



Coordinator

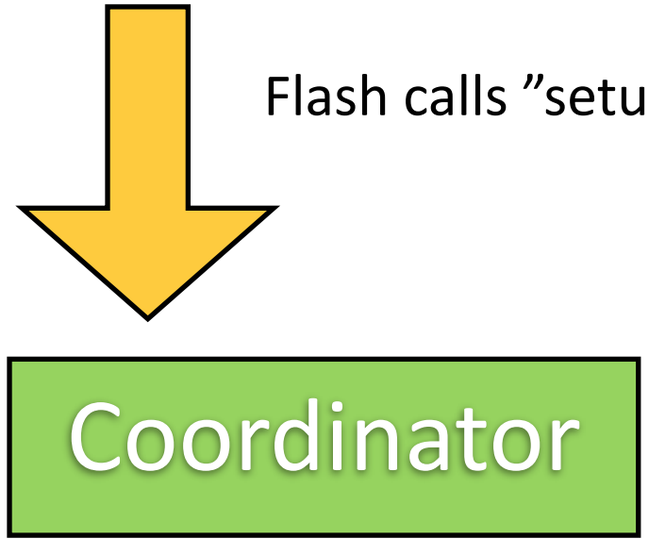
Lock

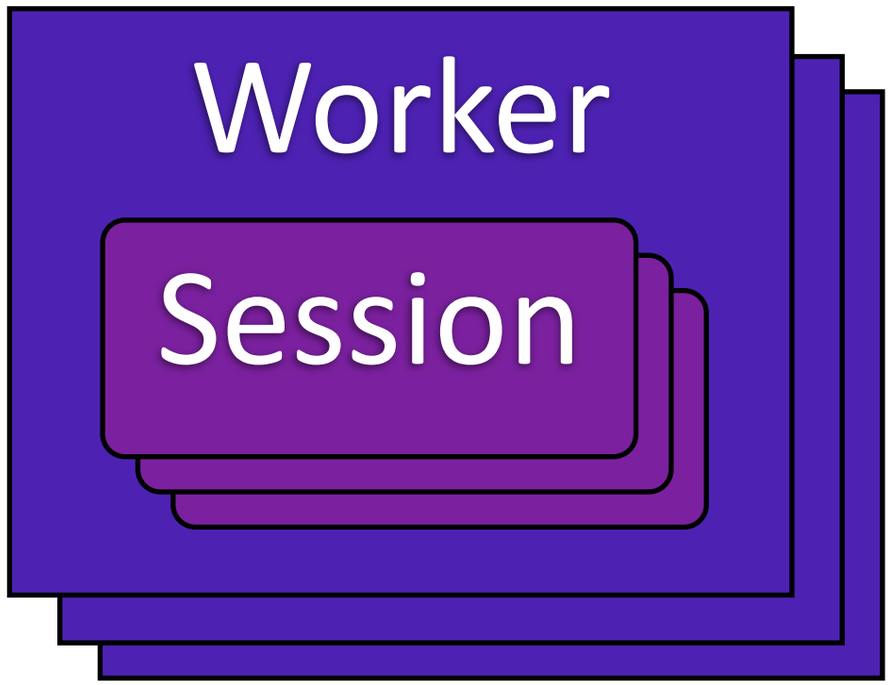
DBs

wooga
world of gaming

New user comes online

Flash calls ”setup”





Lock

DBs

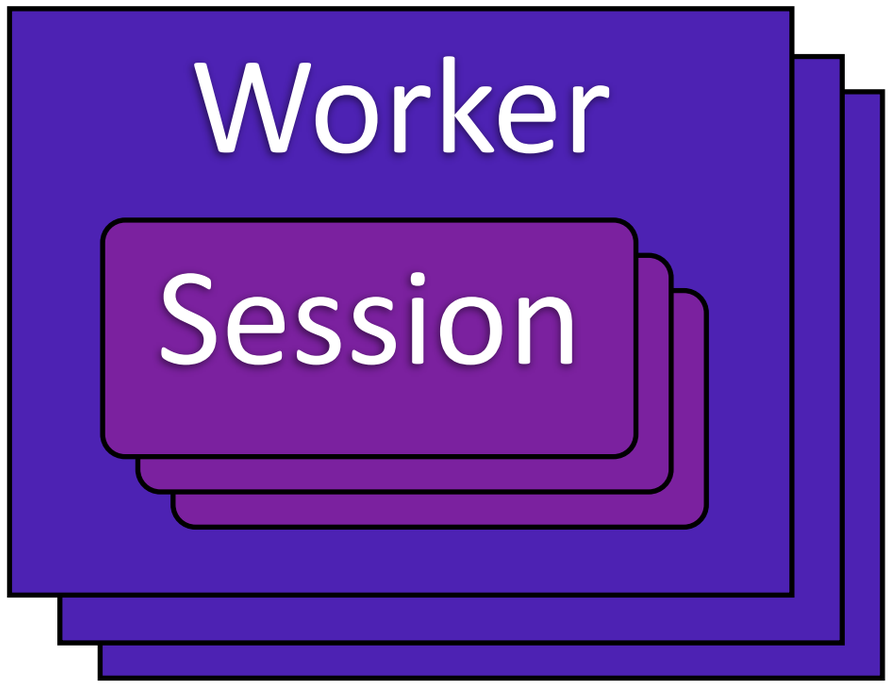
wooga
world of gaming

New user comes online

Flash calls ”setup”

Coordinator

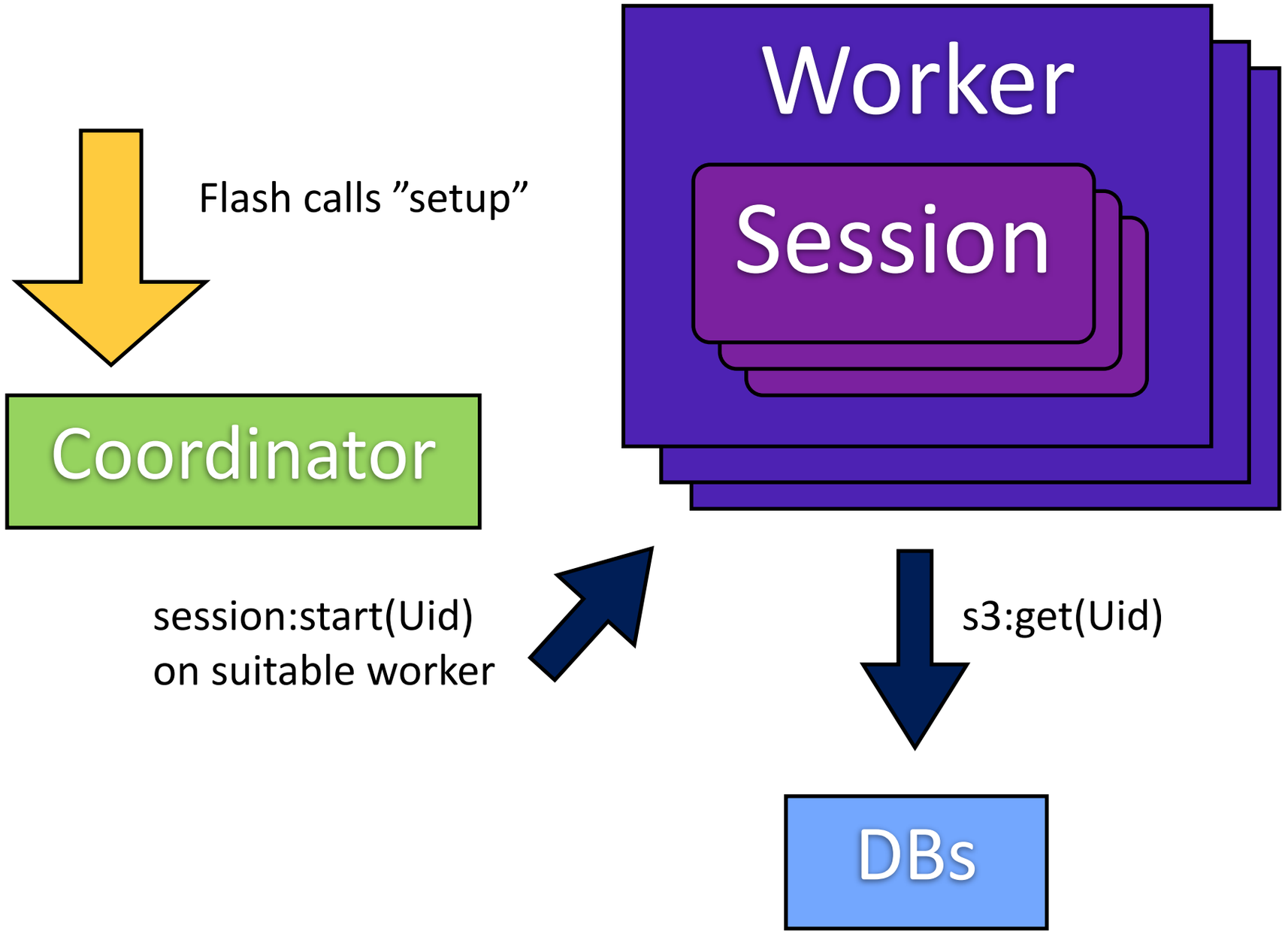
session:start(Uid)
on suitable worker



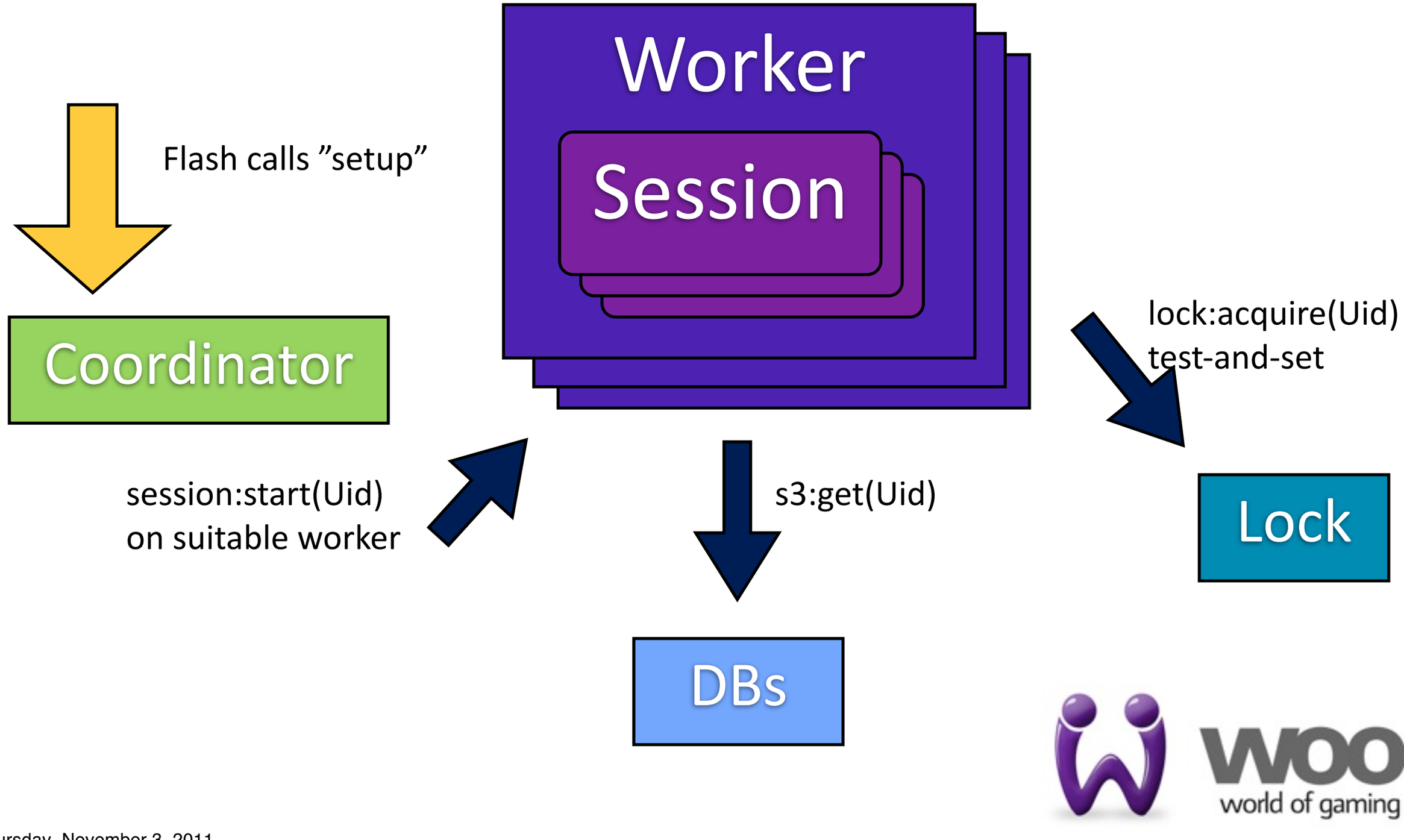
Lock

DBs

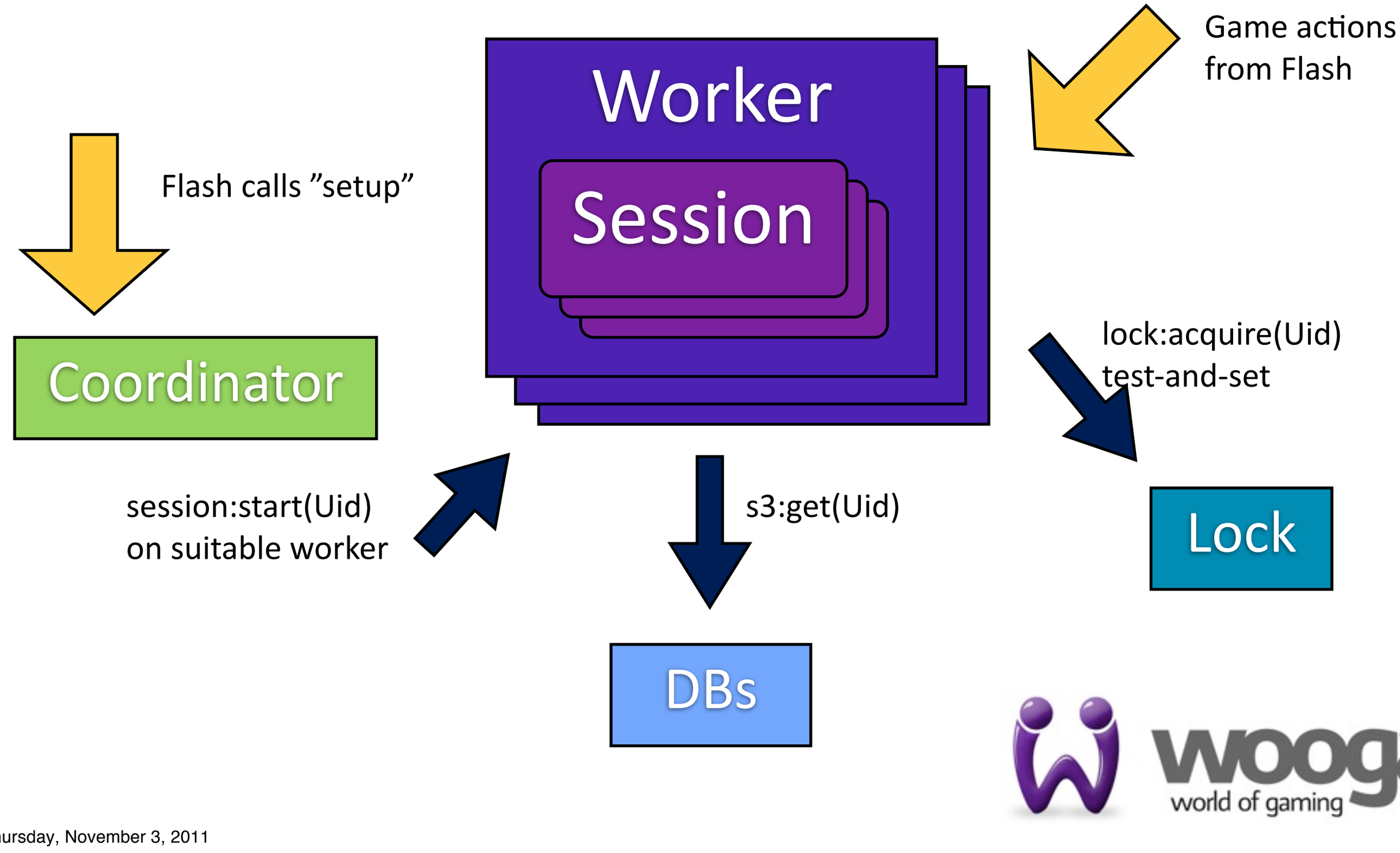
New user comes online



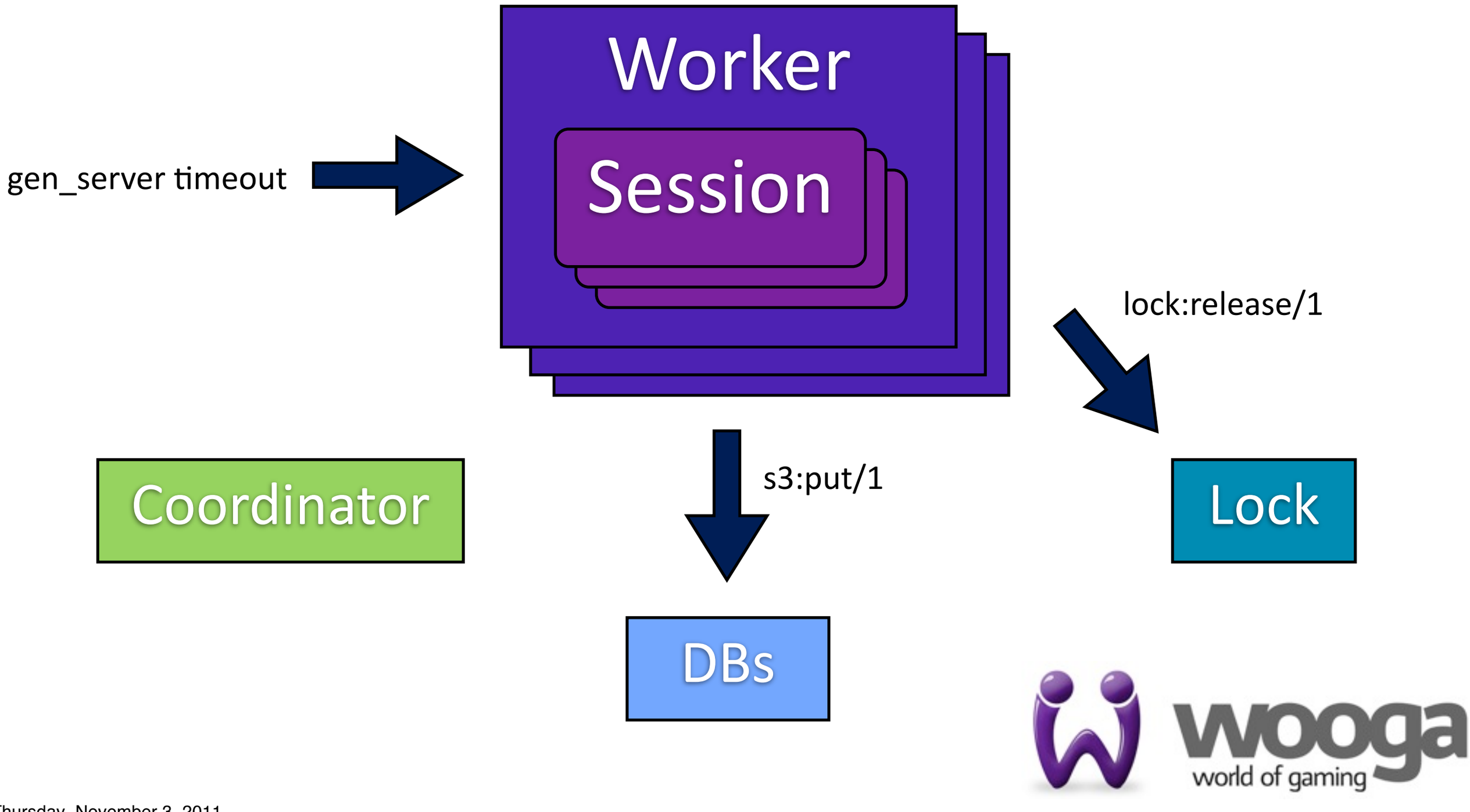
New user comes online



New user comes online



User goes offline (session times out)



Implementation of game logic

Dream game logic

Fast

+

Safe

Dream game logic

Dream game logic

- We want high throughput (for scaling)
  - Try to spend as little CPU time as possible
  - Avoid heavy computation
  - Try to avoid creating garbage
- ..and simple and testable logic (correctness)
  - Functional game logic makes thinking about code easy
  - Single entry point, gets ”request” and game state
  - Code for happy case, roll back on game-related exception

How we avoid using CPU

- Remove need for DB serialization by storing data in process
- Game is designed to avoid heavy lifting in the backend, very simple game logic
- Optimize hot parts on the critical path, like collision detection, regrowing of forest
- Generate erlang modules for read-heavy configuration (~1 billion reads/1 write per week)
- Use NIFs for parsing JSON (jiffy)

wooga
world of gaming

How to find where CPU is used

- Profile (eprof, fprof, kprof[1])
- Measure garbage collection (process_info/1, gcprof[2])
- Conduct experiment: change code, measure, repeat
- Is the increased performance worth the increase in complexity?
- Sometimes a radically different approach is needed..

[1]: github.com/knutin/kprof

[2]: github.com/knutin/gcprof

wooga
world of gaming

Operations
August 2011 - ...

Operations

- At wooga, developers also operate the game
- Most developers are ex-sysadmins
- Simple tools:
  - remsh for deployments, maintenance, debugging
  - automation with chef
  - syslog, tail, cut, awk, grep
  - verbose crash logs (SASL)
  - alarms only when something *really* bad happens

Deployments

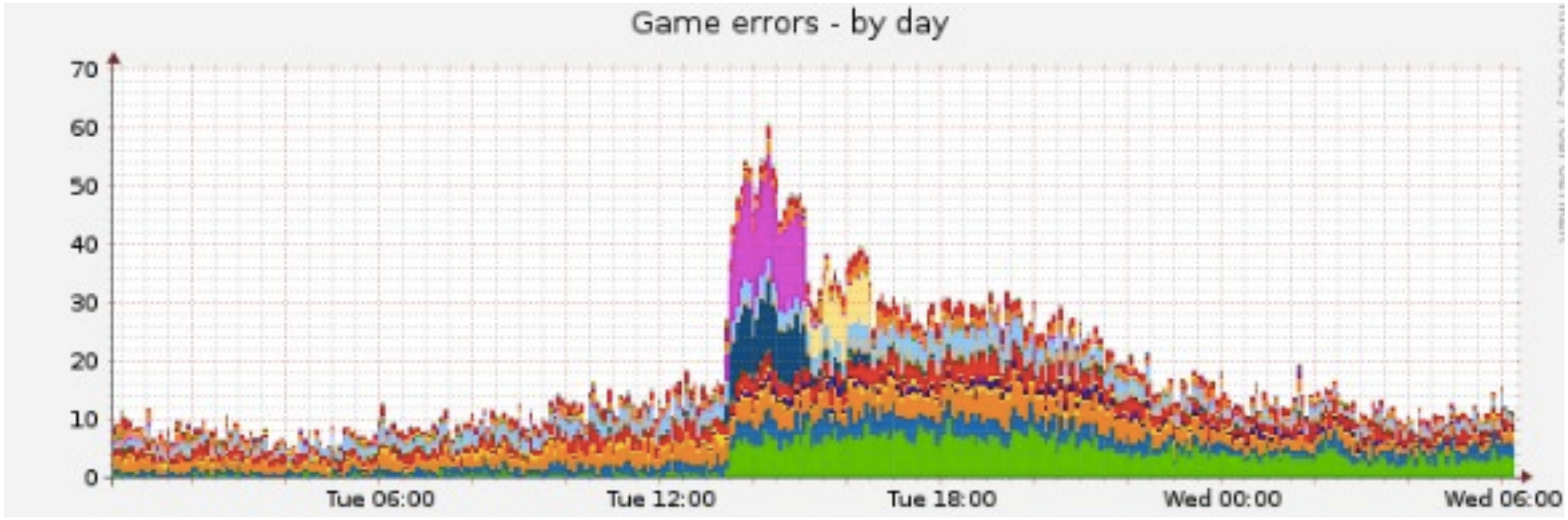
- Goal: upgrade without annoying users
- Soft purge
- Set system quiet (webserver & coordinator)
- Reload
- Open the flood gates
- Migrate process memory state on-demand
- Total time not answering game requests: < 1s

wooga
world of gaming

How we know what's going on

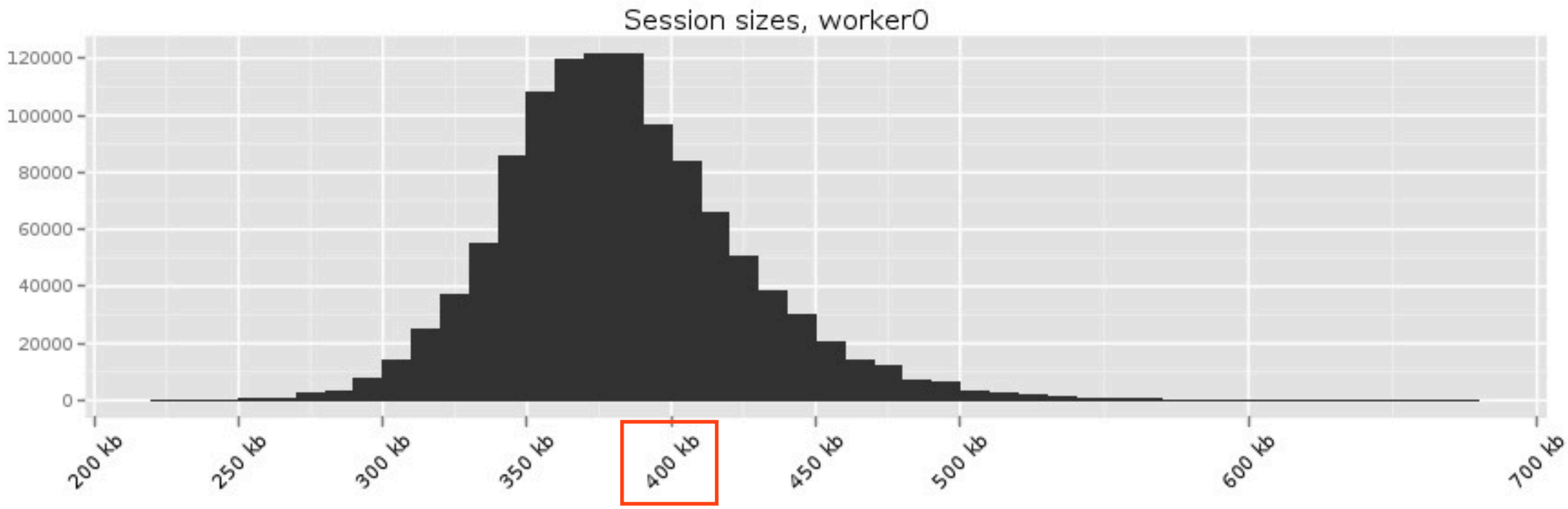
- Event logging to syslog
  - Session start, session end (process memory, gc, game stats)
  - Game-related exceptions
- Latency measurement within the app
- Use munin to pull overall server stats
  - CPU and memory usage by beam.smp
  - Memory used by processes, ets tables, etc
  - Throughput of app, dbs
  - Throughput of coordinator, workers, lock

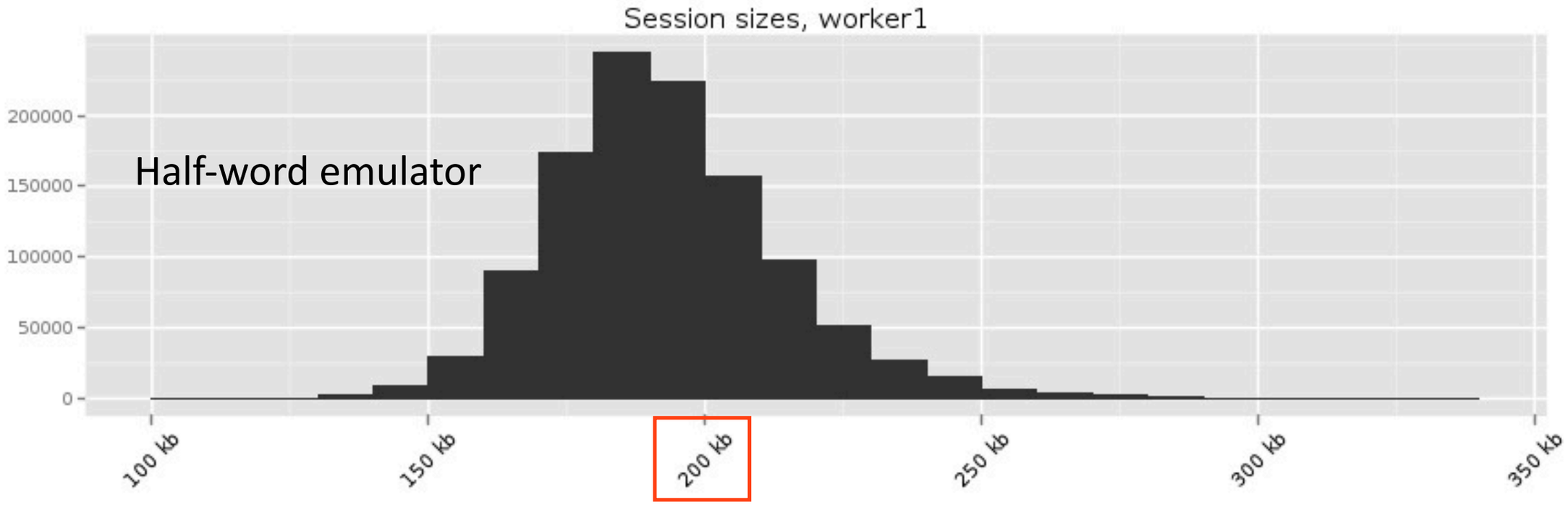
How we know what’s going on



Game error: The game action is not allowed with the current server state and configuration

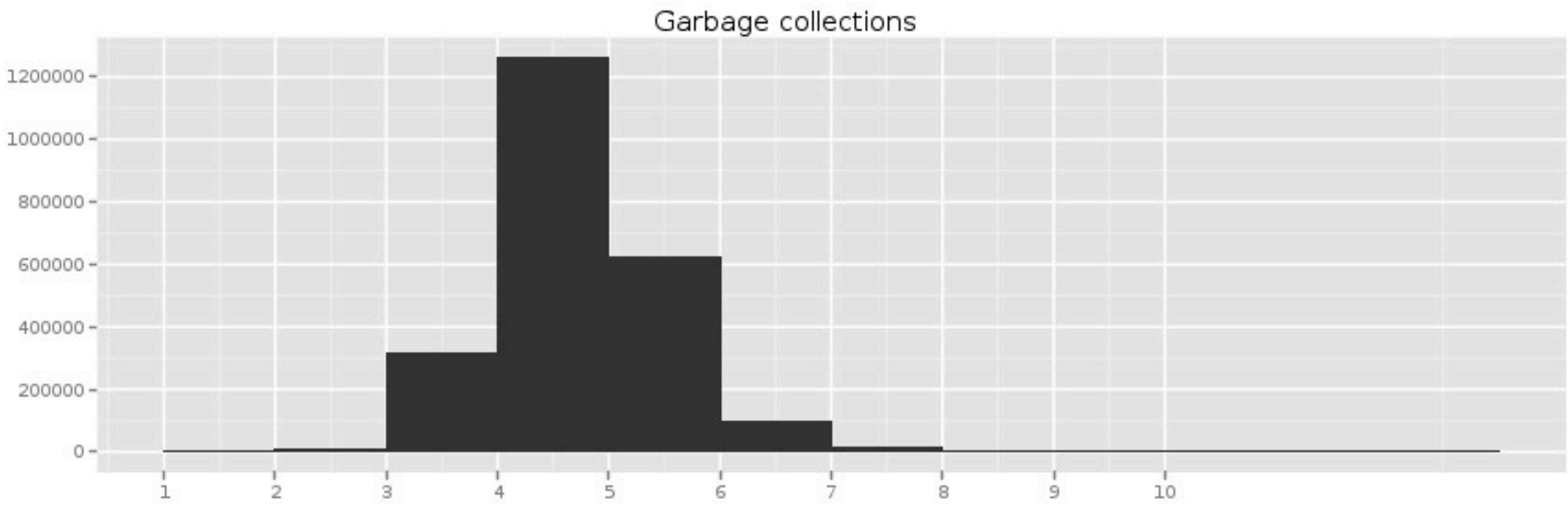
How we know what's going on



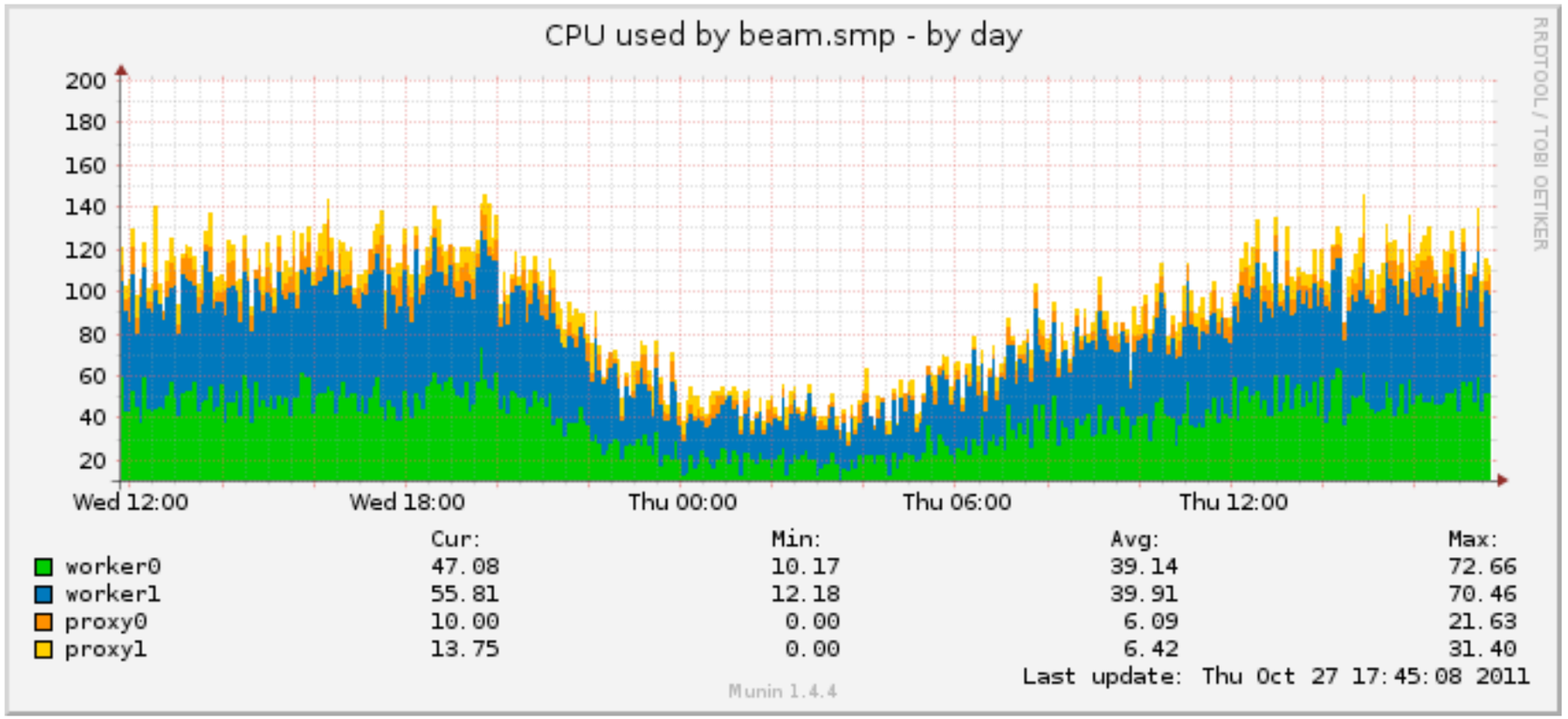


Half-word emulator

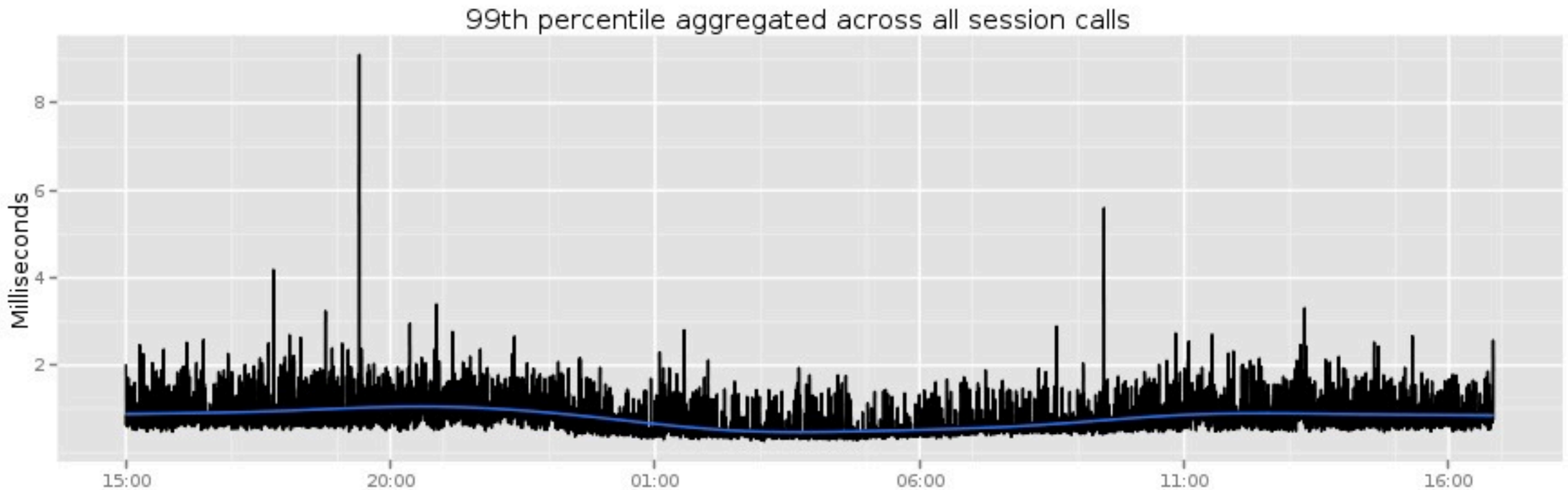
How we know what’s going on



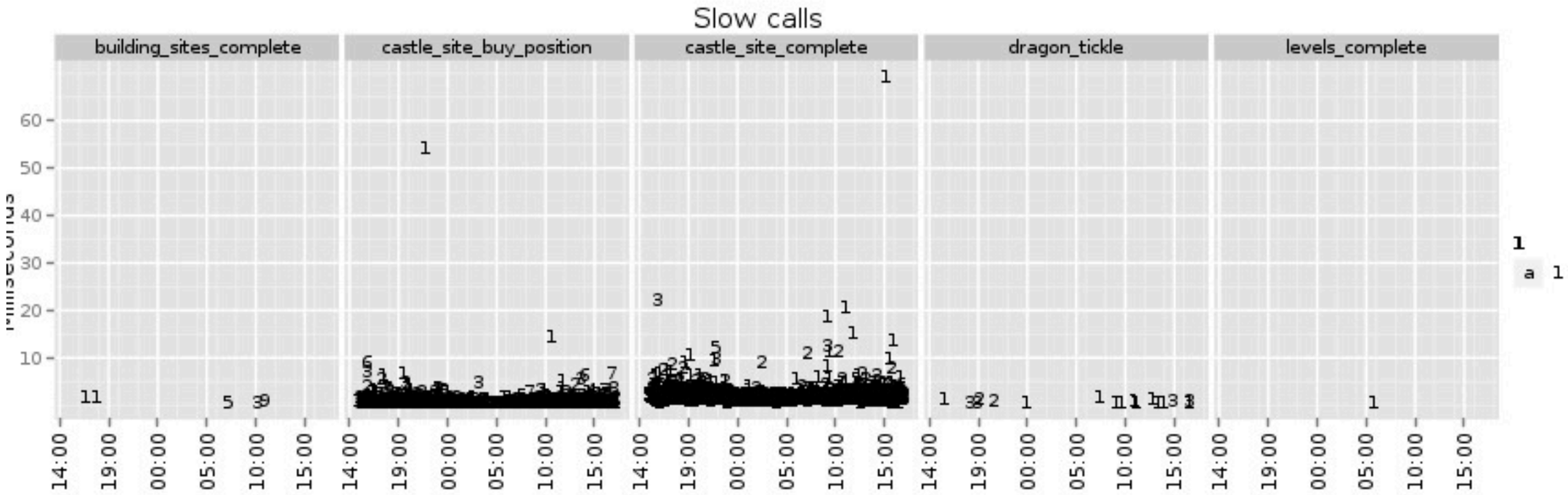
How we know what's going on



How we know what’s going on

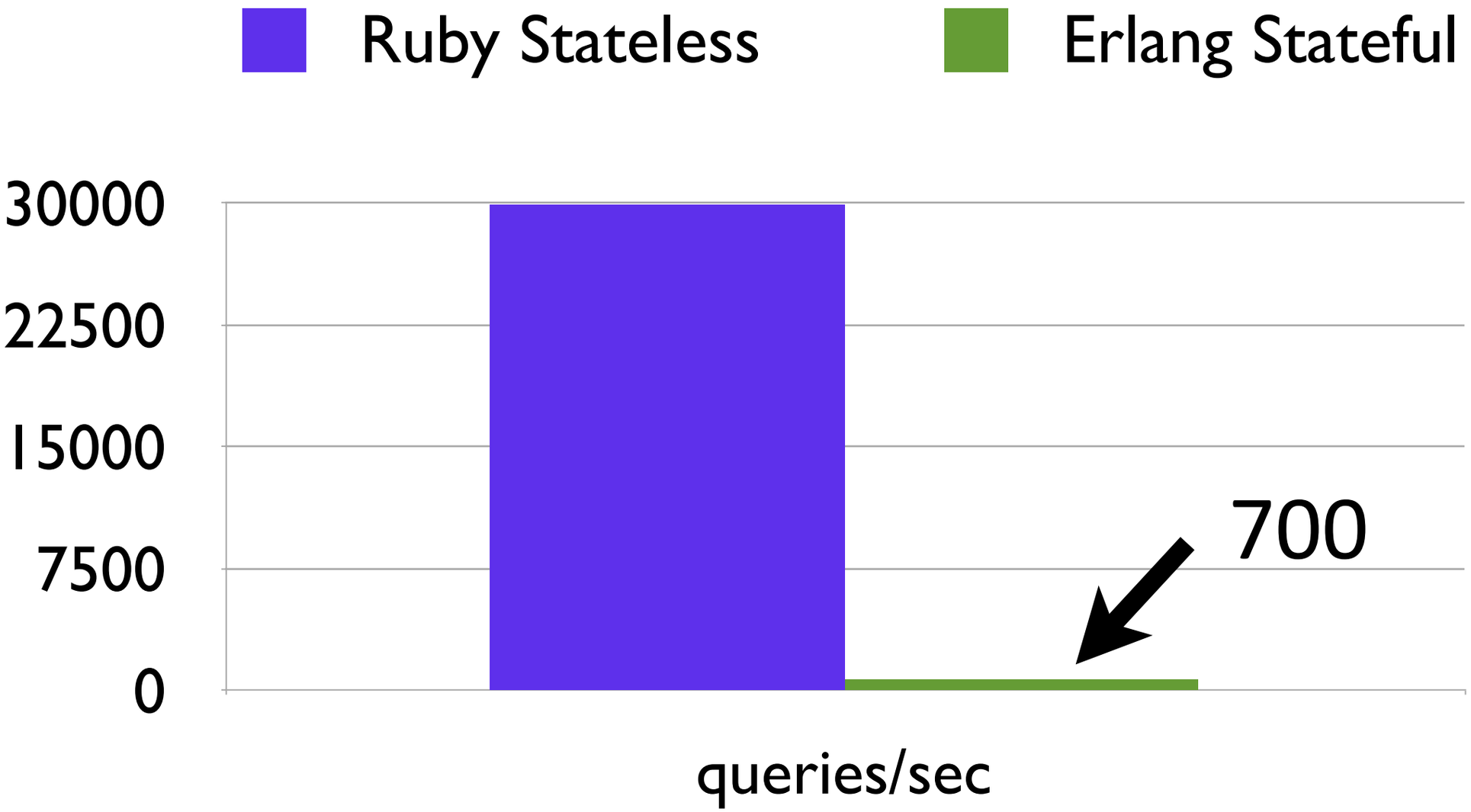


How we know what's going on



Conclusions

Conclusions - database



Conclusions
Db maintenace

AWS S3 as a main datastore

one document/user

0 maintenance/setup cost

Redis for the derived data (leaderboard etc.)

load very far from Redis max capacity

Conclusions
data locality

- average game call < 1ms
- no db roundtrip at every request
- no need for low latency network
- efficient setup for cloud environment

Conclusions
data locality

- finally CPU bound
- no CPU time for serializing/deserializing data from db
- CPU only busy transforming data (minimum possible activity)

Conclusions
CPU usage

- 300K daily users
- 1000 http req/sec (game actions)
- 4 m1.large AWS instances (dual core 8GB RAM)
- 2 instances (coordinators) 5% CPU load
- 2 instances (workers) 20% CPU load

wooga
world of gaming

Conclusions
extra benefits

entire user state in one process

+

immutability

=

Transactional behavior

Conclusions
extra benefits

One user session -> one erlang process

The erlang VM is aware of processes

=>

the erlang VM is aware of user sessions

Conclusions

Thanks to VM process introspection

process reductions -> cost of a game action

process used memory -> memory used by session

We gained a lot of knowledge about a fundamental ”business” entity

Conclusions

- a radical change was made possible by a radically different tool (erlang)

Conclusions

- a radical change was made possible by a radically different tool (erlang)
- erlang can be good for data intensive/high throughput applications

Conclusions

- a radical change was made possible by a radically different tool (erlang)
- erlang can be good for data intensive/high throughput applications
- stateful is not necessarily hard/dangerous/ unmaintainable

Conclusions

- november 2010: 0 lines of erlang @wooga
- november 2011: 1 erlang game server live

...with more erlang coming, join us

Q&A

Knut Nesheim @knutin

Paolo Negri @hungryblank

http://wooga.com/jobs