Real-Time Performance at Massive Scale

Fredrik Linder
Machine Zone, Inc.

MachineZone

Machine Zone

Machine Zone delivers highly engaging, social, real-time multi-player games for the mobile market

We are a top grossing mobile game company

Global Reach

- One global world
- 5M-40M+ daily active users
- 100k-500k concurrent users
- N client platforms
  - iOS
  - Android
  - Windows
- 24 x 7 x 366

Global

MachineZone

Social 2.0

- Defining Social 2.0
  - **No language barrier**
  - Global scale and reach
- Social World Changer
  - Not just social gaming

Social
2.0

Massive Scaling

- Big Scale Architecture
  - Scaling up + out
  - Fast drives, fast network
  - Memory is cheap
- Fully Distributed
- Fully Redundant
- Cluster Native
  - Automatic failover
- Cloud Aware

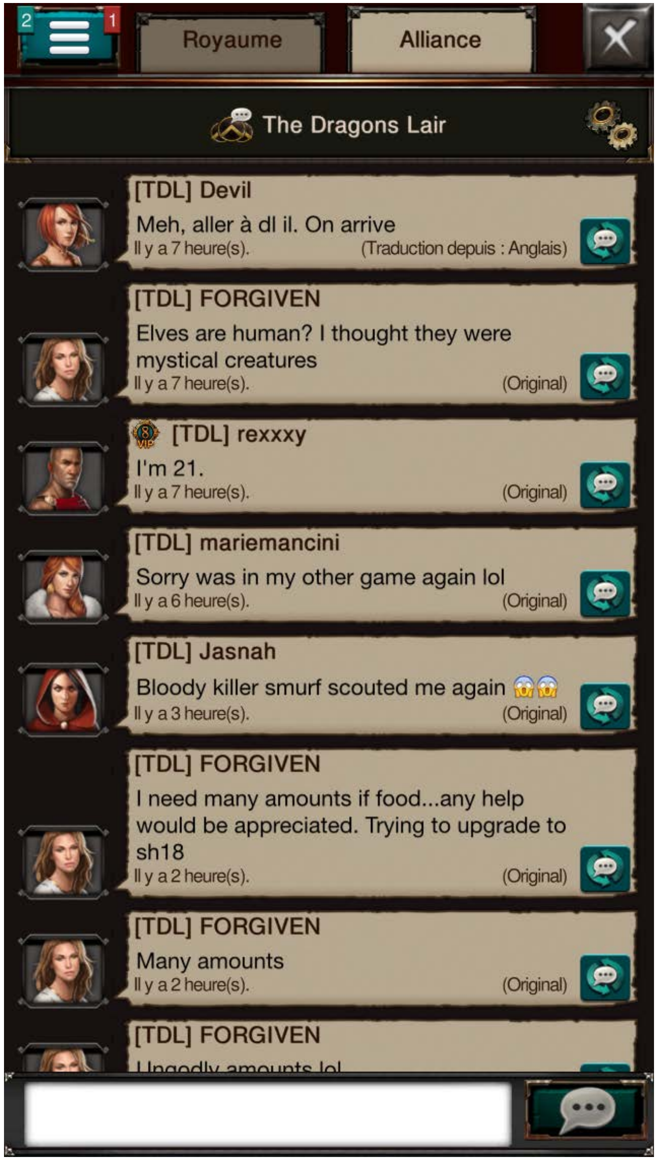
Massive Scaling

What we use Erlang for

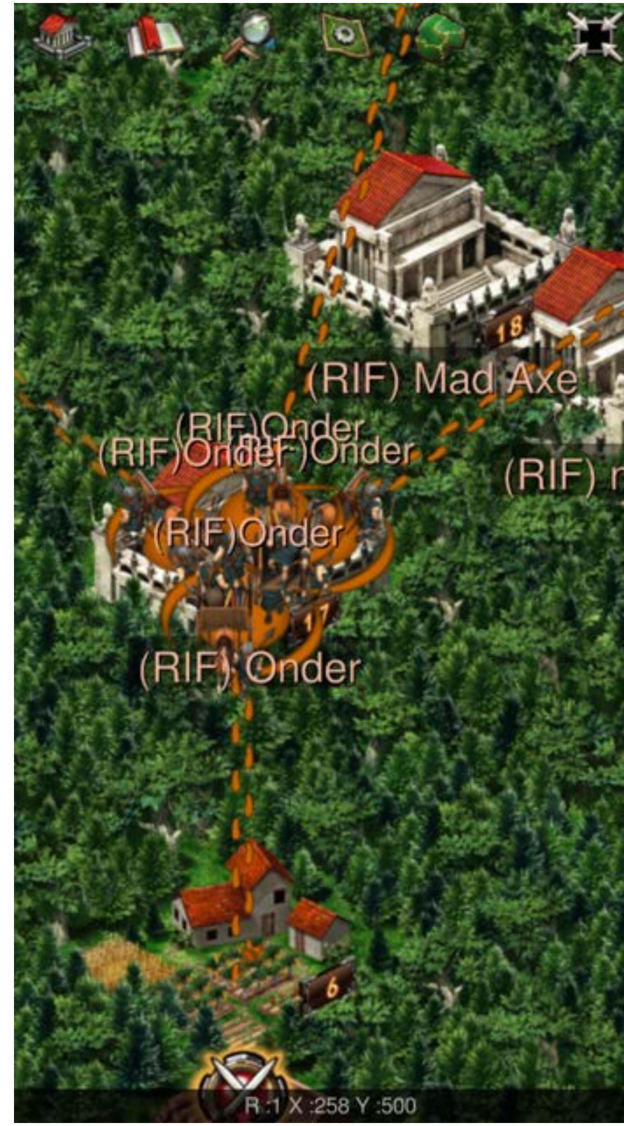
- Real-time world updates
- Real-time events
- Real-time timers
- Real-time 1-1 chat
- Real-time group chat
- Real-time translations
- Real-time event processing
- Real-time notifications

Real-Time

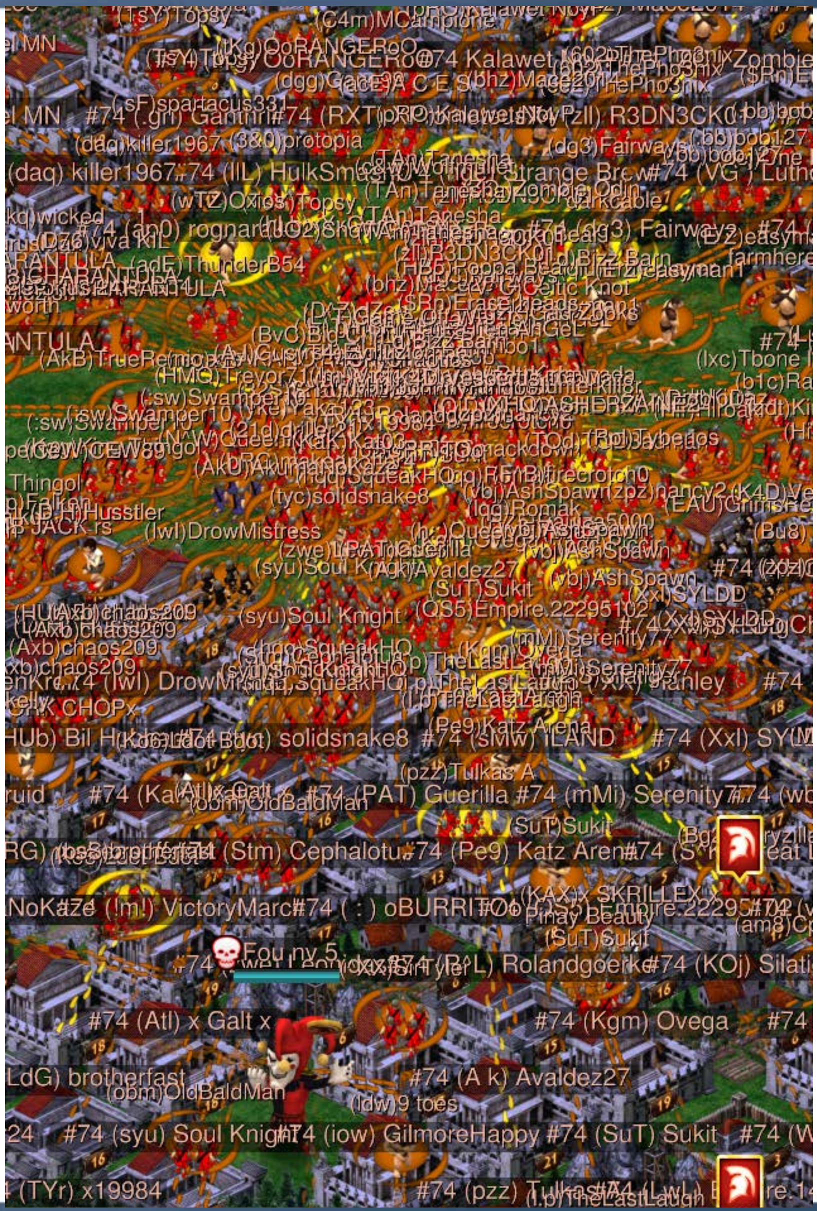
Real-Time Translation @ Massive Scale



Real-Time @ Massive Scale




#2 (sic) KINGSLAY3R
(NJI)GingerBytch
m)MrGees 2
#2 (NJA) CrazyZwirko
#2 (NJA) Legat


#74 (Atl) x Galt x
#74 (Kgm) Ovega
LdG) brotherfast
#74 (A k) Avaldez27
#74 (syu) Soul Knight
(TYr) x19984

Demo

Challenges

- Architecture support
- Everyone
  - On the same map
  - Has the same view of game state
  - Can communicate with anyone else
  - No language barrier
- Must feel natural
  - Real-time (soft)
- Downtime
  - less revenue
  - less users

Our approach

"If it doesn't scale we can't use it"

"Have performance goals"

"Always have a fallback"

1. Measure early
   – Understand and predict growth, operational issues and user behavior
2. Benchmark, stress test, failover testing
   – Identify bottlenecks, ensure we can meet capacity needs before releasing
3. Iterate and improve

MachineZone

An example: Real-time event processing

```
handle_info(#info{payload=Payload},State) ->
    {Worker,State2}=pick_worker(State),
    worker:handle_payload(Worker,Payload),
    {noreply,State2};
```

Iteration 1 – Baseline

- 3rd party pool lib
  - One dispatcher process
  - Pool of workers
- Queue = dispatcher and worker inboxes
  - Inbox lost on process crash
  - Dead worker may receive new msgs
    - LocalPid ! Msg

Iteration 1 – Goals

- No message loss!
- Push-based
- Processes should crash on failures
- Fast
- Linear scalability
- Option to persist queued messages

Iteration 1 - Solution

- A few short iterations later:

- Prioritize control messages
- Adding a NIF queue for traffic data
  - Lock-less multi-producer-multi-consumer
  - Optionally mmap:ed to disk
- Owned by a separate process
- Workers pop one msg at a time
- Off-line retries

MachineZone

Iteration 1 - Numbers

- NIF queue
  - half as fast as erlang:send(LocalPid,Msg)
- Low contention
  - atomic operations
- Scales linearly, but:
  - No timeout argument in NIF call

On to other things

Iteration 2 – Goals

- Need a similar solution in another project
- Need back-pressure
- Need to persist queue off-host
- Need more QoS options
- Need to detect failing worker node

Iteration 2 – Reuse

- Broke out NIF queue + pooling into separate lib
  - A NIF is always a risk
- New requirements superseded old ones
  - Backpressure > Speed
  - Involved processes may live on different nodes
  - Workers should still have a single payload at a time
  - Queue better live on the Erlang side

Iteration 2 – Solution

```
[client]    [dispatcher]    [queue]       [worker]
   :             :             :             :
   +-call(Msg)-->|             :             :
   |             +-call(Msg)-->|             :
   |             |<- - -queued-|             :
   |<- - -queued-|             |-call(Msg)-->|
   :             :             |<- - ongoing-|
   :             :             :             |
   :             :             :           {work}
   :             :             :             |
   :             :             |<-call(done)-|
   :             :             |-ok- - - - ->|
   :             :             :             :
```

Iteration 2 – Numbers

- 11µs per request w/ 2 workers
- 11µs per request w/ 8 workers
- 11µs per request w/ 32 workers
- 13µs per request w/ 256 workers
- 17µs per request w/ 1k workers
- 31µs per request w/ 4k workers
- 90µs per request w/ 16k workers

Summary

- We knew what we needed to build
- Solve scaling first
- Measure and benchmark as part of dev process

- I hope to open source soon

MachineZone