Optimizing Distributed Actor Systems for Dynamic Interactive Services

Andrew Newell (Purdue University), Gabriel Kliot (Google), Ishai Menache (Microsoft), Aditya Gopalan (Indian Institute of Science), Soramichi Akiyama (University of Tokyo), and Mark Silberstein (Technion)
Dynamic, state changes at runtime
Interactive, users demand fast responses
Run at large scale
Example Service

Interactive, users actively chatting

Dynamic, state is always changing

Millions of users

Chatroom

Other examples

- Social networks
- Online gaming
- Internet of Things
Actor Model

- State kept by actors
- Upon receiving message
  - Update state
  - Send message to actors
  - Create actors

Scaling to millions of users
- CPU to handle requests
- Memory to store state

User clients
- Bob
- Sue
- Jon

Server

Actors
Distributed Actor Systems

- Examples: Orleans, Erlang, Akka
- Eliminate cost of development at scale
  - Add enough servers to handle load
  - Fault tolerance and correctness
- Latency suffers
Challenge: Low Latency at Scale

- Inter-server: messaging overhead
- Intra-server: resource allocation
- Scaling dynamic interactive services
Outline

- Inter-server messaging problem & solution
- Intra-server resource allocation problem & solution
- Evaluation on Orleans
At scale, many messages cross server boundaries
Remote Messaging Overhead

Profile of request latency

- Worker: 32%
- Receiver: 32%
- Sender: 25%
- Other: 10%
- Network: 1%

Over half of latency is due to inter-server message processing

Goal, reduce remote messaging with better actor placement

Typical workload on multiple Orleans servers
Random placement

Colocation placement

Load balancing
Remote messaging

First call

Static workload remote messaging
Load balancing
Dynamic workload remote messaging

Remote messaging always high on dynamic workloads
Actor Placement Model

- Balanced graph partitioning
  - Vertices: actors
  - Edge weights: messaging
  - Partitions: servers

- Messaging graphs
  - Reasonable partition exists
  - Dynamically changes

- Cost constraints
  - Scales with actors and servers
  - Minimize actor movements
1. Decentrally find a good partner
   1. 1 swap at a time
   2. Cooldown timer
2. Perform swap protocol
   1. Improve balance
   2. Reduce messaging
3. Repeat
Swapping Protocol

1. A identifies and sends candidate actor set to B
2. B selects candidate set
3. B picks swapping subsets
   1. Improve balance
   2. Reduce remote messaging
4. B responds with swap decision
Outline

- Inter-server messaging problem & solution
- Intra-server resource allocation problem & solution
- Evaluation on Orleans
Orleans Runtime Overview

Staged Event Driven Architecture (SEDA)

Individual thread pools

Orleans default: thread per core per stage

How to allocate threads and does it matter?
Thread Allocation Affects Latency

64 different thread allocation runs, average latency collected

Orleans default thread allocation

3x reduction by reducing to just enough threads per stage

Too few threads has huge repercussions

Worker threads

Sender threads
Dynamic Thread Allocation

- Existing work
  - Allocate, check, repeat
- Our solution
  - Measure and directly find global optimum among all stages

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{\lambda_{tot}} \sum_{i=1}^{K} \frac{\lambda_i}{\mu_i - \lambda_i} + \eta \sum_{i=1}^{K} t_i \\
\text{subject to} & \quad \mu_i \geq \lambda_i, \quad \forall i = 1, \ldots, K, \\
& \quad s_i t_i = \mu_i, \quad \forall i = 1, \ldots, K, \\
& \quad \sum_{i} t_i \beta_i \leq p.
\end{align*}
\]

Threads per stage
Outline

- Inter-server messaging problem & solution
- Intra-server resource allocation problem & solution
- Evaluation on Orleans
Evaluation

- Implemented in Orleans
  - Distributed balanced graph partitioning
  - Dynamic thread allocation

- Mimic production Halo Presence workload
  - Maintains stats of players in real-time
  - Clients query for stats of all players in some game
  - Both dynamic and interactive
Workload

Dynamically changing at runtime
- Players start playing
- Players move between games
- Players stop playing

Queries for stats of a game

10 Orleans servers
100k players
12.5k games
Remote Messaging %

- Random placement, 90%
- Converges at 12%

<10 minutes to go from 90% -> 12%

Actor Movements Per Minute

Time (minutes)
Thread Allocation

At 90% remote messaging

Receive 1 thread  Worker 5 threads  Send 2 threads

At 12% remote messaging

Receive 1 thread  Worker 6 threads  Send 1 thread
Latency Results

- **Median latency (ms)**
  - Baseline: $40\%$ reduction
  - Actor Partitioning: $20\%$ reduction
  - w/ Thread Allocation

- **99th percentile latency (ms)**
  - Baseline: $70\%$ reduction
  - Actor Partitioning: $20\%$ reduction
  - w/ Thread Allocation
Conclusion

- Demonstrated latency problems and solutions for distributed actor systems
  - Actor placement
  - Thread management (in Orleans’s open source)
- Better resource management in distributed actor systems
  - Easy to develop at scale
  - Low latency -> interactive services
- Techniques can apply beyond actor systems
Orleans, open source
https://github.com/dotnet/orleans

E-mails
- andyjnewell@gmail.com
- gkliot@google.com
- ishai@microsoft.com
- aditya@ece.iisc.ernet.in
- s.akiyama@aist.go.jp
- mark@ee.technion.ac.il