Optimizing Distributed Actor Systems for Dynamic Interactive Services

Andrew Newell *
Purdue University
newella@purdue.edu

Gabriel Kliot*
Google
gkliot@google.com

Ishai Menache
Microsoft Research
ishai@microsoft.com

Aditya Gopalan
Indian Institute of Science
aditya@ece.iisc.ernet.in

Soramichi Akiyama*
University of Tokyo
s.akiyama@aist.go.jp

Mark Silberstein
Technion
mark@ee.technion.ac.il

Abstract

Distributed actor systems are widely used for developing interactive scalable cloud services, such as social networks and on-line games. By modeling an application as a dynamic set of lightweight communicating "actors", developers can easily build complex distributed applications, while the underlying runtime system deals with low-level complexities of a distributed environment.

We present *ActOp* — a data-driven, application-independent runtime mechanism for optimizing end-to-end service latency of actor-based distributed applications. *ActOp* targets the two dominant factors affecting latency: the overhead of remote inter-actor communications across servers, and the intra-server queuing delay. *ActOp* automatically identifies frequently communicating actors and migrates them to the same server transparently to the running application. The migration decisions are driven by a novel scalable distributed graph partitioning algorithm which does not rely on a single server to store the whole communication graph, thereby enabling efficient actor placement even for applications with rapidly changing graphs (e.g., chat services). Further, each server autonomously reduces the queuing delay by learning an internal queuing model and configuring threads according to instantaneous request rate and application demands.

We prototype *ActOp* by integrating it with *Orleans* – a popular open-source actor system [4, 13]. Experiments with realistic workloads show latency improvements of up to 75% for the 99th percentile, up to 63% for the mean, with up to $2\times$ increase in peak system throughput.

* Work done while at Microsoft Research.

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

Distributed actor systems [1, 3, 4] are a natural fit for building online applications with numerous dynamically interacting objects, like social networks, on-line games and Internet of Things applications [8, 11]. The actor model was proposed decades ago to represent concurrent systems as a set of interacting *actors*, and recently has seen increasing interest as a convenient framework for developing complex cloud services. Modern actor systems simplify the design by mapping application objects onto lightweight actors which encapsulate object state and logic, and dynamically interact via asynchronous messages. For example, in an online chat service, every user and chat room can be modeled as an actor. Importantly, actor systems boost developer productivity by providing convenient concurrency control, as well as built-in runtime support for a variety of functionality essential for developing distributed systems, e.g., Remote Procedure Calls (RPC) and fault tolerance.

Although actor systems proved convenient for building complex online services, scaling them to sustain increasing service demand while maintaining low-latency response remains difficult. As the number of actors grows, more servers are added to the system to accommodate the load. Assigning new actors to servers in a way that simultaneously balances the load across the servers and ensures low service latency poses a challenge. Widely used placement policies, e.g., in key-value store systems, employ consistent hashing or random assignment to determine a server for each object (actor, in our case), which maintains the number of objects-per-server roughly equal [16, 17, 32]. However, in contrast to passive objects in key-value stores, actors *interact*. Thus, a hash-based or random placement policy might assign such

interacting actors to different servers, which in turn results in a client request traversing multiple servers. As we show in our experiments using the *Orleans* distributed actor system [4], such remote actor interaction is detrimental to end-to-end service latency (see §3).

Unfortunately, traditional actor systems like Akka [1] and Erlang [3] include no mechanisms for automatic system performance optimization, e.g., via locality-aware actor assignment to servers. They provide numerous hooks and configuration options that enable such low-level optimizations, but leave the implementation to the application developer. For example, in an online chat service, placing heavily communicating actors on the same server, like a chat room actor and all its user actors, helps reduce service latency significantly (as we show in §3). Yet, it is the developer's responsibility to identify which actors communicate, how to optimally assign them to servers, and how to migrate actor instances to their new physical locations. Moreover, with the inherently dynamic nature of object interaction in such applications, e.g., joining new users or new chat rooms, the graph of inter-actor communications is constantly and unpredictably changing. As a result, achieving low latency requires the developers to implement continuous runtime monitoring and complex application-specific mechanisms, complicating the design significantly.

In this paper we design ***ActOp***, a principled, application-independent mechanism for dynamic performance optimization in scale-out distributed actor systems. *ActOp*'s techniques can be implemented by application developers in actor-based services as a part of the application logic. Yet, they are primarily designed to be integrated into an actor system *runtime*, to monitor and automatically adapt to changing application characteristics transparently to the application. Therefore, we prototype *ActOp* by integrating it into the runtime of *Orleans* – an open-source actor system used in production by many cloud-based online services [5].

Our primary goal is to optimize locality of interacting actors, co-locating them on the same server to reduce the end-to-end service latency. We model the actor-to-server assignment problem as a *balanced* partitioning of the actor interaction graph. The solution simultaneously minimizes communications between the servers while equally distributing load among them. We design a novel, fully distributed graph-partitioning algorithm which scales to millions of actors. The algorithm is specifically tailored to distributed applications with rapidly changing actor interaction graphs. It avoids the communication bottleneck of centralized solutions, in which all actors report to a single controller node that stores the whole interaction graph. Instead, each server actively monitors local actors, and performs pairwise exchanges of updates with other servers, thereby eliminating centralized coordination and state accumulation entirely. The algorithm is running continuously and migrates actors across servers transparently and unobtrusively for the application.

While actor locality optimization improves the service latency in a distributed system as a whole, our profiling shows unexpected performance degradation of individual servers when migration is enabled. We find that the root cause lies in significant fluctuations in the server workload, caused by shifting actors from one server to another, and as a result, suboptimal thread allocation in each server. Specifically, the design of many high-concurrency servers in general [7, 10, 16], and *Orleans* in particular, follows a Staged Execution Design Architecture (SEDA) [33]. In *SEDA*, the server internal logic is represented as a set of stages, each with its own thread pool and task queue. *SEDA* proved extremely useful for achieving stable throughput scaling under load. However, the performance is highly sensitive to the number of threads in each *SEDA* stage. When actors are migrated to improve locality, the internal server load shifts from send/receive stages to application logic stages, calling for dynamic reallocation of the threads among them to improve server performance.

We design a mechanism that dynamically optimizes the allocation of threads to *SEDA* stages. Unlike the previous heuristic solutions [33], we formulate and analytically solve the service latency minimization problem using a queuing model of a *SEDA*-based *Orleans* server. We obtain a closed-form solution that allows to compute the number of threads in each stage as a simple function of the queuing model parameters, e.g., the arrival rate to each stage; this enables low-overhead tuning of the thread allocation at runtime. *Orleans* is instrumented to monitor and record these parameters in the actual running system, dynamically adjusting the thread allocation to match the current system load.

We evaluate *ActOp* using two realistic services implemented on top of *Orleans* with *ActOp* : a heartbeat service and an online gaming service with up to a million actors running on multiple servers and serving thousands of requests/second. Our distributed graph-partitioning algorithm reduces the 99th percentile latency by up to 69% (from 736ms in the original system to 256ms with the optimization), and the mean latency by up to 56% (from 41ms to 21ms). Such an improvement in the end-to-end service latency is crucial for providing satisfactory user experience in online gaming systems. In a single server with optimally allocated threads, the 99th percentile latency is reduced by up to 68% and the mean latency by up to 58%. Both optimizations together achieve even larger reductions, up to 75% for the 99th percentile, and up to 63% for the mean.

Our optimizations improve both latency and throughput by enabling a more efficient use of system resources. Specifically, reducing the amount of communication between remote objects decreases the object serialization overhead, and allocating an optimal number of threads in a server reduces the OS thread management overhead. The end result is a significant reduction in the CPU utilization, which not only improves response time, but also doubles peak system throughput. In summary, our contributions are as follows:

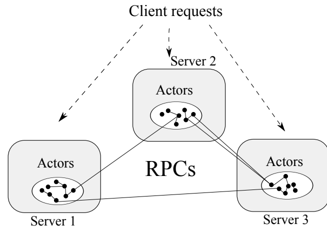


Figure 1. An actor communication graph in *Orleans* distributed actor system.

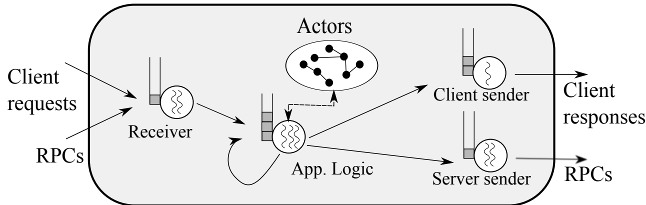


Figure 2. SEDA architecture of a server in *Orleans*.

- We identify excessive remote inter-actor communication and unoptimized intra-server thread allocation as key factors affecting latency in actor-based interactive services (§3).
- We address these issues by formulating a dynamic graph partitioning problem, and a parameter optimization problem in a queuing network. We design a novel distributed algorithm for the former (§4), and offer an analytical closed-form solution to the latter (§5).
- We integrate *ActOp* into the *Orleans* actor system and show significant improvements in latency and throughput in realistic distributed applications with up to one million actors under high load (§6).

2. Background

In this section we briefly describe the actor programming model and *Orleans* middleware for distributed actors, highlighting the main aspects that we optimize in this work.

Orleans is an open-source .NET-based distributed runtime based on the actor model [4, 13]. *Orleans* extends the functionality of traditional actor systems like Akka and Erlang by introducing a novel concept of actor virtualization. The system automatically instantiates actors on demand, and fully manages their lifecycle, eliminating the actor management burden from application developers. Further, the system automatically handles hardware or software failures by re-instantiating the failed actor upon the next call to it.

Actor's physical location is hidden from the application, so the runtime may seamlessly migrate actors across servers while the application is running. Further, actor isolation implies that actors cannot uncontrollably share data, allowing the runtime to redistribute threads between actors in a transparent way. By raising the level of abstraction, *Orleans* makes it possible for the middleware to optimize the system performance transparently to the application, a property which we leverage in this work. Similar ideas are applied in other actor systems, e.g., Orbit [9] and Azure Reliable Actors [6].

Applications based on actor systems. Developers write applications in a familiar object-oriented programming style, and *Orleans* turns each object into a stateful actor. Method calls to an actor on the same machine are executed as Local Procedure Calls (LPC), which involves deep copying of the arguments to provide full isolation between actors. Calls to an actor on a remote machine are automatically translated into Remote Procedure Calls (RPC) via seamless serialization/deserialization of arguments and return values. Figure 1 illustrates a communication graph of actors in a distributed application that spans multiple servers.

Actor partitioning and scale out. Actors are instantiated and placed randomly on different servers to allow scaling in terms of request load and number of actors. *Orleans* allows to migrate active actors while rerouting RPCs among servers. Similar mechanisms are employed in other large scale systems (e.g., Dynamo [16], WAS [15]).

Server architecture. Each server in *Orleans* handles a large number of concurrent messages received and sent by the actors it hosts. *Orleans*'s programming model utilizes user-level threads: every actor executes on one user-level thread (*Task* in C#). These logical threads are multiplexed on a small number of physical threads that execute all application logic [13]. *Orleans*'s internal design follows the well-known Staged Event Driven Architecture (SEDA) [33] – a popular design choice for high-concurrency servers, like Apache Camel [7], SwiftMQ [10] and Dynamo [16]. SEDA represents request processing as a pipeline of fine-grained stages, with a separate task queue and a fixed thread pool for every stage. In *Orleans* there are three main stages (see Fig. 2): receive message (including de-serialization), execute application logic (physical threads executing logical threads), and send message (including serialization). Carefully allocating threads to each stage is important for server performance, as we elaborate below.

3. Performance Overheads in *Orleans*

We analyze the performance of a typical actor-model based distributed application running on *Orleans* in order to identify the main factors causing latency degradation.

Halo Presence is an interactive application which implements presence services for a multi-player game running in production on top of *Orleans*. The service allows players to a) join an existing game, b) discover other players, c) watch the game live while receiving periodic updates on the game progress. The service resembles a chat service: games are chat rooms, and the players periodically broadcast the events that occur in their view of the game. There are two types of actors: games and players.

Orleans makes it easy to scale the system to thousands of games in which millions of players interact in a dynamic

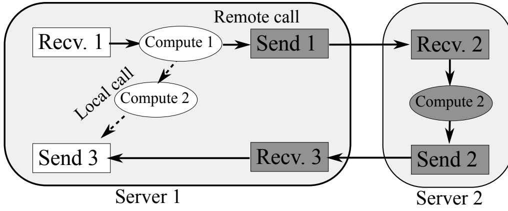


Figure 3. Execution flow of a request that serially accesses two actors when actors are on the same server (white shapes) and on different servers (dark shapes).

way. It instantiates actors on demand, distributing the load across multiple servers transparently to the application. Programmers are oblivious to the physical location of the actors and use standard function calls to invoke functions of both remote and local actors.

Scaling the *Halo* Presence service, however, introduces an additional challenge of achieving low latency, because the service is interactive. Remote clients (real players) query the server to find out the status of the other players in the game, and slower response significantly affects user experience. Building mechanisms to achieve high throughput and low latency while scaling to millions of actors is the primary motivation of this work.

Our measurements show that the original *Orleans* system fails to satisfy low latency even at a moderate scale. Running on ten 8-core CPU servers, using 100K concurrent live players with eight players per game, and a total of 6K client requests/second (80% of CPU utilization per server), the median, 95th and 99th percentile end-to-end latency of a single request is 41msec, 450msec and 736msec respectively (we further elaborate on our experimental settings in §6). Such high latency is unacceptable as it gives the perception of a "sluggish" service. In the following we show that remote messaging due to the lack of locality between interacting actors significantly impacts the service latency.

Optimizing actor locality is essential for improving service latency. Each server sustains on average 600 client requests per second at 80% CPU load, which seems low given the I/O bound nature of the service requests. However, the total actual number of requests handled by each server is substantially higher, and is dominated by the actor-to-actor communications between the servers.

In *Halo* presence, each player actor sends one message to its game actor that broadcasts it to the 8 players, who in turn respond back. Thus, each client request results in 18 additional messages sent across the actors. Since *Orleans* assigns actors to servers at random to avoid hotspots, the vast majority of actor-to-actor interactions cross server boundaries. This is consistent with our finding that $\approx$ 90% of all messages between actors are remote. Consequently, the number of requests handled by each server is more than an order of magnitude higher than the number of external client requests.

Accessing remote actors is an essential part of any distributed actor-based middleware, however it incurs significant latency and server load overheads. To understand why, we compare the (simplified) execution flow of a remote actor call (dark shapes) and a local actor call (white shapes) in Figure 3. A remote call involves several costly extra stages: after the server processes the request (Compute 1), it serializes the arguments and forwards the request to another server (Send 1), which deserializes (Recv. 2), processes (Compute 2), and serializes and sends the results back (Send 2). A local call does not require the serialization/deserialization steps and invokes the compute stage directly (enqueues a request for execution for Compute 2). Consequently, request latency decreases and effective server throughput increases if the requests are handled only by local actors.

To evaluate the potential benefits of actor locality for system performance, we invoke the same workload with 100K concurrent players, but now with most of the communicating player actors co-located on the same server. We observe that the median, 95th and 99th percentile latency reduce to 24msec, 100msec and 225msec respectively – a significant improvement in system performance.

Static actor assignment is insufficient. The *Halo* presence application is a classic example of a modern social application with highly dynamic actor interaction graph. Clients may leave and join games, and engage in interaction with other clients located on other servers at arbitrary times. Therefore, any static actor placement policy which optimizes the placement for a particular actor interaction graph would eventually lead to a similar amount of remote interactions (such as the random policy evaluated here), since the communication graph changes and the static initial placement becomes invalid after some time.

None of the static built-in policies in *Orleans* is sufficient to achieve both load balancing and communication locality. Consider, for example, a *local placement* policy, in which an actor is instantiated and placed on the server where it was first called. Subsequent calls from other, potentially remote actors, will be performed as RPC calls. This policy works well when the callee actor is exclusively owned by the caller actor, however it fails to capture many other scenarios. For example, if the first caller happens to rarely interact with the callee later, collocating them on the same server results in excessive network communications because subsequent, more frequent callers may reside on other servers.

Another notable disadvantage of the local placement policy is that it might lead to a skewed and unbalanced actor distribution across servers, which is an impediment to system scale up. Thus, *Orleans* is by default configured with a simple *random placement* policy, which chooses the server for a new actor uniformly at random. While this policy forgoes the actor locality, it achieves good load balance and throughput scaling.

Server thread allocation must be optimized. The importance of thread allocation in SEDA-based high-concurrency servers and its impact on system throughput was studied in

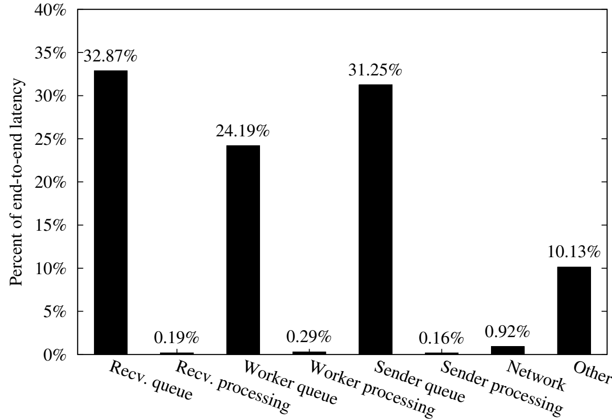


Figure 4. Average latency breakdown for a single request: times in queues, processing time in each stage, and network latency. "Other" includes OS queuing and other factors.

Worker Threads \ Sender Threads	2	3	4	5	6	7	8
8	30.7	31.7	30.8	34.8	38.2	37.2	32.3
7	24.9	26.9	30.7	30.7	31.4	36.6	32.4
6	25.4	24.5	23.6	25.7	25.6	25.2	28.5
5	19.1	18.6	20.4	20.4	23.1	23.7	23.7
4	16.8	15.8	18.7	18.1	18.6	19.1	18.8
3	11.9	12.7	14	14.7	15.8	15.1	15.5
2	13.2	9.9	11.4	10.4	12.1	13.2	12.9

Figure 5. Server request latency (msec) with different thread allocations.

the past [33, 34]. Here we present experimental results showing the significance of thread allocation for service latency. We analyze the lifespan of a client request on a single *Orleans* server. We run a simple counter application where in response to a client request an actor increments a counter. We invoke 15K requests/sec on 8K actors. Figure 4 shows the average latency a request spends in SEDA stages and queues described in §2. We use the default thread allocation policy in *Orleans* : a thread per stage per CPU core. We profile a request from the moment it arrives until it leaves the server.

Queuing delay dominates the end-to-end latency, and by far exceeds the network latency, as well as processing time in each stage. This is a symptom of an incorrect allocation of threads across the stages. Indeed, Figure 5 shows a heat map of the median service latency under different thread allocations with the same workload. In the figure, *workers* refer to the application threads, *senders* refer to the threads performing serialization and sending of the results. Thread allocation policy drastically affects the end-to-end latency: the worst-performing allocation (8 workers, 6 senders) results in 4× higher latency than the best-performing allocation (2 workers, 3 senders). Notably, the default configuration used in *Orleans* (8 workers, 8 senders for an 8-core machine in our case) is among the worst-performing configurations.

Combining thread allocation and locality optimization is necessary to achieve high performance. Dynamically optimizing actor locality and thread allocation together has a greater effect on the end-to-end latency than the effect of each optimization separately. There are two complementary performance benefits of placing communicating actors on a single server. First, doing so will directly reduce the latency of individual client requests that involve multiple remote actors. Unlike local actor-to-actor interaction, remote messages traverse server queues where they suffer from per-message delays. Beyond that, a second latency improvement is expected for all the requests handled by each server, because of fewer remote messages per server, hence lower CPU load.

Unfortunately, this second performance benefit will not materialize if the thread allocation is not dynamically adjusted to the changing server workload. Servers optimized for remote messaging allocate more threads to the send/receive stages; recall that these stages perform serialization/deserialization and therefore are CPU-intensive. As a result of better actor locality and fewer inter-server interactions, these stages become underutilized, while the actor logic execution stage may get overloaded and become the bottleneck. Consequently, the service time of a request in a single server will grow, instead of the expected improvement.

The above analysis leads us to the design of an *online* optimization framework that constantly monitors server performance and actor communications and quickly adapts to the highly dynamic nature of actor-based applications at scale. The scalable actor partitioning and live migration mechanism we describe in §4 strives to optimize the placement at runtime transparently to the application. At each server, dynamic re-allocation of threads to server stages is achieved via a model-driven thread allocation mechanism (§5); the mechanism monitors internal server performance metrics, and reallocates the threads to achieve lower service latency.

4. Locality-Aware Actor Partitioning

We present a mechanism for dynamic actor partitioning with the objective of reducing inter-server communications. We start by formulating the optimization problem. We then describe our distributed algorithm (§4.2), and highlight important implementation details (§4.3).

4.1 Actor assignment as balanced graph partitioning

We consider a system where interacting actors are partitioned across multiple servers. We model such a system as a graph, where each actor is a vertex, and each edge is an interaction between the respective actors. The edge weight reflects the overall cost of interaction, e.g., frequency and amount of transferred data in each interaction.

Our solution needs to simultaneously satisfy the following goals: (1) balanced partitioning of actors across the servers to avoid hot-spots and overloaded servers to provide high

system throughput; (2) co-locating frequently interacting actors on the same server to reduce inter-server communication. We formulate these goals as a balanced graph partitioning problem: *Given n available servers, we seek to partition the graph vertices into n disjoined balanced sets such that the sum of edge weights crossing the partitions is minimized.*

Formally, let V be the set of vertices, and define V_p as the subset of vertices located at server p, i.e., $V_p \subseteq V$ and $V_p \cap V_q = \emptyset$ (we disallow actor replication, so an actor may reside on one server only). We denote by $w_{v,u} \geq 0$ the weight between any pair of vertices v and u; the weight is proportional to the average number of messages sent from v to u (see §4.3 for details on how this average is determined).

The total communication cost C is given by

$$C = \sum_{\{v,u: v \in V_p, u \in V_q, p \neq q\}} w_{v,u}.$$

The goal is to find a new partition $V_1', V_2', ..., V_n'$, $\cup_i V_i = \cup_i V_i'$, such that the total cost C is minimized. As mentioned above, the new partition should be balanced, namely $\big||V_p| - |V_q|\big| \leq \delta$ for any servers p and q, where δ is an imbalance tolerance parameter.

Assumptions. For the sake of presentation we assume that all actors consume a similar amount of memory and compute resources on each server. This assumption is realistic for the applications we consider, because the actors used are usually lightweight and rarely compute-bound (since they require fast response time). Further, because the actors are lightweight and fine-grained, and their per-actor state is small (relative to a state of a whole process, for example), we assume that the overhead of migrating actors to another machine is low, and do not explicitly model it in our optimization. We briefly discuss in §4.2 possible algorithmic extensions that explicitly consider actors of different sizes and migration overheads. Although we do not explicitly incorporate migration costs in our current formulation, we recognize that massive reshuffling of actors across machines is undesirable in real systems. Thus, our algorithm limits the number of migrated actors at any exchange between the servers (see §4.2 for details).

Design alternatives. The main design goal is to allow the partitioning mechanism to support rapidly changing graphs with millions of vertices.

One notable design option is to store and process huge graphs on a single server (e.g., using GraphChi [23] or similar graph processing frameworks). However we rule out such a centralized design primarily due to its poor scaling. Solving the graph partitioning problem exactly for large graphs is prohibitively slow, even when using linear-time heuristics. Our attempts to solve the partitioning problem instances of representative sizes of tens of servers and a few millions of actors (graph nodes) with METIS [20], the well known and de-facto standard library of graph algorithms, required several hours to finish. Moreover, simply collecting all the data in one location while accommodating the high rates of actor graph updates becomes a performance bottleneck and does not scale. For rapidly time-varying actor graphs, e.g., about 1% of all the edges changing every minute as in the *Halo* presence example, the messaging and processing overheads of a centralized solution result in an unacceptable delay, making the outcome of the partitioning algorithm likely to be obsolete by the time it finally becomes available.

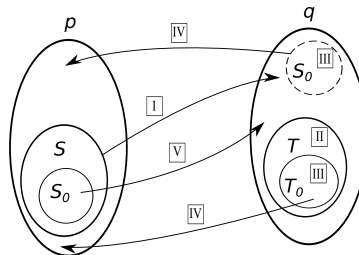


Figure 6. Visualization of the steps for the pairwise coordination protocol detailed in Alg. 1.

On the other hand, we rule out a fully distributed approach in which servers migrate actors unilaterally based on their local views of the actor graph. While this approach would scale for large graphs, it may result in highly unbalanced partitions: a given server might overload neighboring servers because it is oblivious to their actual load that depends also on migrations from their other neighbors. Second, unilateral migrations might by inefficient. For example, a "heavy" communication edge between two actors $u \in p$ and $v \in q$, might trigger p sending u to q and q sending v to p around the same time. We, therefore, seek a solution that is as scalable as the uncoordinated approach, but provides better precision. In this context, we point out that the algorithm we present next is inspired by the distributed solution proposed in [30]. However, there are two fundamental differences. First, the algorithm in [30] operates under static graph conditions. Second, our algorithm employs batching at a server level, while [30] operates on a vertex-by-vertex basis (actors in our case), which limits its scalability for the graph sizes we consider.

4.2 Distributed partitioning algorithm

Our algorithm relies on distributed coordination between *pairs* of servers.

Pairwise coordination protocol. Every server p maintains the list of edges from the vertices of p to other vertices in the system. The pairwise protocol is invoked independently and periodically by each server. In each round, a server p initiates an exchange procedure described in Algorithm 1 and illustrated in Figure 6. p calculates the candidate set $S \subset V_p$ for every server (as described below) and picks the server q with the best candidate set. q (and corresponding S) is chosen according to the cost reduction anticipated by p. q may completely reject the whole exchange operation or accept it partially. It rejects the operation if it has been recently involved in another exchange (in our experiments,

1: Server p sends an exchange request to server q along with a candidate set of actors S
if q exchanged recently **then**
q rejects p's request
else
2: q determines a candidate set T that may be sent to p
3: q determines the subset $S_0 \subset S$ of accepted actors from p, and the subset $T_0 \subset T$ of actors to be transferred to p
4: q transfers subset T_0 to p, and notifies p on accepted set S_0
5: p accepts the subset T_0
end if

Algorithm 1: Pairwise coordination protocol. Server p initiates an exchange request to server q

the exchange is rejected if a previous exchange took place less than a minute ago). If q rejects the exchange, p attempts an exchange with a remote server which would lead to the second best cost reduction, and proceeds until some server accepts the exchange request or until all servers with positive cost reduction reject it.

Assume that q accepts the exchange request. q then inspects the candidate set S, and decides which subset to accept (denoted S_0), and which of its own actors it should send back to p (denoted T_0). We note that q may decide to reject some or even all of the vertices in S, if those do not reduce its own cost. This could happen since the graph may have changed since p collected the information based on which it formed S, or because p operated on a partial sample of the communication graph which produced inaccurate edge estimates. As we elaborate below, the exchange should also adhere to the balancing constraint. We next explain how the candidate sets are chosen, and how the remote server q determines the exchange subsets $S_0 \subset S$ and $T_0 \subset T$.

Determining the candidate set. For every local vertex v and remote server q, p calculates a *transfer score* $R_{p,q}(v)$, which is the cost reduction expected in p from migrating v to q. The transfer score is computed as the sum of weights of all v's edges that used to be remote (and will be local after migration) minus all the edges that used to be local (and will be remote after migration). Formally, $R_{p,q}(v) = \sum_{u \in V_q} w_{v,u} - \sum_{u \in V_p} w_{v,u}$. p maintains a candidate set of k vertices with the highest $R_{p,q}$ found so far. The candidate set contains a small fraction of the total number of vertices in p, which corresponds to limiting the number of actor migrations as explained in Section 4.1. After all transfer scores are computed, p picks the server that has the highest total transfer score which is the sum of the transfer scores of all the vertices in the candidate set.

Determining exchange subsets. Recall that q receives a set of candidate actors S from p. From this point on, it is q who will make the decisions: which subset $S_0 \subset S$ to accept, and which subset of actors T_0 it will transfer to p. First, q picks a candidate set T of its own actors which can be potentially transferred to p, in the same way as p picks S, so that at this stage q still ignores the potential consequences of taking actors from S. After T is determined, q has to determine $S_0 \subset S$ and $T_0 \subset T$. This is a balanced graph-partitioning problem, which is known to be NP-hard. While logarithmic approximations to the problem do exist, e.g., [12, 24, 29], the underlying algorithms are difficult to use in practice, as they require solving an LP/SDP or huge packing problems.

Instead, we design an iterative greedy procedure which jointly determines S_0 and T_0 while adhering to the balancing constraint. Initially q constructs two sorted max-heaps, such that each heap stores the transfer scores of the vertices in S and T respectively. In each iteration step, q chooses the candidate vertex with the highest transfer score among all vertices. If the vertex migration would violate the balance constraint between q and p, the algorithm chooses the highest-scored vertex from the other heap. The selected vertex v is marked for migration and removed from the respective heap. q then updates the scores of all the remaining vertices in both heaps to reflect the migration of v. The algorithm proceeds until both heaps are empty or no more vertices can be moved due to the balancing constraint.

Computational complexity. Let V be an upper bound on the number of vertices in any server; recall that k is the size of the candidate set, and n is the total number of servers. Straightforward analysis yields that the running complexity of each pairwise interaction is $O(nV \log k + k^2)$. The dominant factor is V, implying that the complexity is practically linear in the number of vertices in the server. We note that V becomes effectively much smaller than the total number of vertices once edge-sampling is incorporated (§4.3), making the protocol tractable in practice.

Stability. We next provide a basic stability result for our algorithm for a static communication graph.

Theorem 1. *Let $G = (V, E, W)$ be any weighted graph. Alg. 1 (applied on G) converges to a locally optimal partition*[1] *after finitely many executions, with probability one. Moreover, the resulting partition satisfies $||V_p| - |V_q|| \leq \delta$ for every pair of servers p, q.*

The theorem follows by showing that the overall communication cost decreases monotonically with every migration. Indeed, if a vertex v is chosen to be migrated, it is only because its transfer score s_v is positive. We omit a detailed proof for brevity. Note that convergence cannot be theoretically guaranteed when the graph is dynamic and practical adjustments are incorporated to the algorithm (such as edge sampling). Nevertheless, the above result indicates that the algorithm targets a low-cost and balanced solution. In our ex-

[1] A locally optimal partition of $G = (V, E, W)$ is a partition of the vertices into servers, i.e., $\cup_p V_p = V$, $V_p \cap V_q = \emptyset \; \forall \, p \neq q$, such that for each pair of servers p, q: every vertex in $V_p \cup V_q$ either has a negative pairwise transfer score, or has a positive pairwise transfer score but moving it to the other server violates the balance constraint between p and q.

periments, we empirically show that the algorithm is indeed stable (Fig. 10(a)).

Discussion. The algorithm is designed to operate using only a partial and potentially outdated communication graph available at each server. We also considered a simpler version of the algorithm which performs only one-sided updates, namely, in every iteration a server migrates the vertices with the highest transfer score without any coordination. In our experience, however, this simple algorithm converges much slower, and results in higher inter-server communication and higher imbalance between servers. Therefore, we introduce the pairwise interaction to speed up convergence, and make both servers exchange their high-score vertices while maintaining the load balancing constraint.

As mentioned earlier, our algorithm can be extended to accommodate different actor sizes and migration overheads, as follows. To explicitly consider migration costs we add a term to the transfer score $R_{p,q}(v)$, which is inversely proportional to the actor size. Similarly, we limit the size of the candidate set by the sum of sizes of all actors, and accordingly set the imbalance tolerance δ to represent the allowed imbalance in total size (rather than in number of actors). The evaluation of these extensions is outside the scope of this paper.

4.3 Implementation details

Edge sampling. We target very large graphs with millions of vertices and tens of millions of edges. Although it may be feasible to store the entire graph on a single server, e.g., using an efficient sparse format [23], storing and calculating all edge statistics in our context is unnecessary. The important observation here is that "light" edges would not contribute to the final partitioning, since our algorithm exchanges only small candidate sets in order to minimize communication and computation overheads. Therefore, we store only the "heaviest" edges. Specifically, every server p maintains a list of heavy edges from the vertices of p to other vertices in the system. This is a partial list that includes edges with large weights, which are determined by using the Space-Saving sampling algorithm [26]. Space-Saving is a stream sampling algorithm, and we apply it to the stream of edges observed by a server. The server maintains a constant-size list of the top heaviest edges, which is sufficient to execute our partitioning algorithm.

Gathering edge statistics. We initially used a global concurrent data-structure to maintain communication statistics for each edge. This approach led to a significant latency overhead due to lock contention. Instead, we keep the relevant counters *locally* at each actor, and periodically update the global graph data-structure by traversing all the actors from a single thread.

Transparent actor migration. *Orleans* has a number of useful features which simplify the implementation of actor migration. First, actors in *Orleans* have no notion of location, therefore from the user perspective the migration is transparent. Second, *Orleans* provides an efficient activation/deactivation mechanism which maintains actor's state across activations between different servers. We take advantage of these features in our implementation as follows.

Suppose we migrate actor A from server p to server q. p deactivates A by updating a distributed placement directory. The new placement is now driven by the subsequent messages to A, as follows. Both p and q track in their own location cache that actor A should be placed on server q. If the next message to A comes from p or q, those servers check their location cache and place the actor on q. Otherwise, it will be placed on the server which originated the call. Since the decision to migrate A to q was based on the fact that a high percentage of messages to A came from q, it is likely that the next message to A will originate from q. Intuitively, we probabilistically guarantee that A is placed in the "right" server. This working assumption is verified in our experiments (§6). This opportunistic migration allows us to avoid global coordination which comes with significant overhead. Old cached location values are evicted in order to maintain low space overhead at each server.

5. Latency-Optimized Thread Allocation

We consider a server design that follows *SEDA*, as described in §2. Our goal is to minimize the server response time by determining the number of threads in each *SEDA* stage as a function of load and processing characteristics. We seek a low-overhead solution that allows to quickly recompute the allocation in response to time-varying workload patterns.

5.1 Design alternative: queue-length based control

Previous work on *SEDA*-based servers introduced a queuing-theoretic model of the system [33, 34]. However, we are not aware of any work that directly uses a latency-related formulation to optimize thread allocation. References [33, 34] use the length of the queue for each *SEDA* stage to estimate the stage load. They dynamically allocating more threads to stages with long queues (defined with respect to a queue-length threshold) and de-allocate threads from stages with empty queues. This approach may work well in some scenarios, but has some drawbacks. First, it does not define how to set the queue thresholds, which turns out particularly challenging under the need to support diverse applications and workload characteristics in our framework. More importantly, it is prone to allocation fluctuations, because queue lengths respond in an extremely non-linear fashion to addition of capacity via threads, depending on how close the load is to the capacity, as we elaborate below. The main problem is that even with minor (±1) fluctuations in the thread allocation, the service latency may increase dramatically, e.g., increasing by over 35% when shifting from 2 worker threads to 3 (see Figure 5).

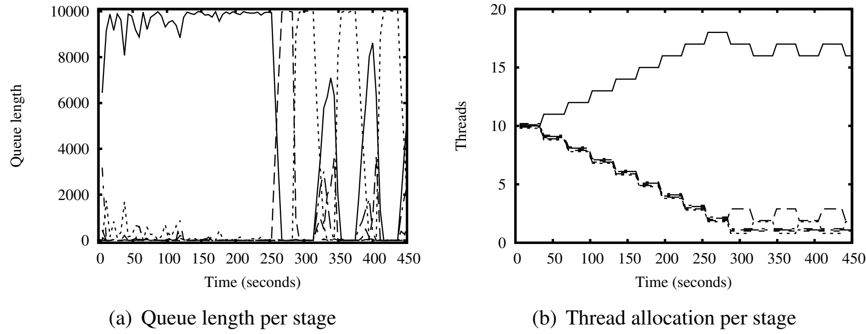


Figure 7. Six stage SEDA with threads allocated based on queue lengths.

To further investigate the queue-based approach, we build a *SEDA* emulator with 6 stages and conduct a thread allocation experiment based on queue lengths. Our queue-based controller samples the queue of each stage every 30 seconds. Any stage with a queue-length greater than T_h requests has its threads increased by one, while every stage with a queue-length less than T_l requests has its threads decreased by one (never going below one thread). T_h and T_l are configurable parameters of the controller. We note that this queue-length based controller is similar to the one proposed in [34].

Figure 7(a) shows the results with $T_h = 100$ and $T_l = 10$. Most stages have virtually empty queues, while the queues of bottleneck stages grow until the queue-length threshold is reached. At this point thread allocations change and as a result the queues flip, as shown in Figure 7(b). Overall, we observe significant fluctuations in queue lengths and thread allocation. Experiments with different values for T_h and T_l yield similar behavior.

This behavior can be intuitively explained using queuing theory. In the M/M/1 queue model, the average queue length is $\frac{\rho}{1-\rho}$. ρ is defined as the arrival rate divided by the service rate, and represents the proportion of available resources that a stage has. When ρ is small (service rate is high, many free threads), differences in ρ do not affect the average queue length much: the queue is almost empty. Only when ρ approaches 1.0 (arrival rate approaches service rate, not enough threads) the queue length grows dramatically. The function $\frac{\rho}{1-\rho}$ is nonlinear in ρ (hence nonlinear in the number of threads), and becomes steeper as ρ approaches one. Consequently, it is hard to smoothly control queue lengths. We can indeed observe from Figure 7 that small changes in the number of threads cause large fluctuations in queue lengths.

Unlike this simple iterative heuristic approach, we choose to leverage the queuing model *directly* to formally define a latency minimization problem parametrized by the actual system parameters we measure at runtime. We are interested in a robust solution which will *simultaneously* determine the thread allocation for all stages, hence will be less prone to allocation fluctuations.

We find a closed-form solution (§5.3) and use it to periodically re-compute the latency optimal thread allocation as a function of instantaneous system load. Doing so poses the challenge of inferring the values of model parameters which are not directly measurable. We describe how we estimate these values in §5.4.

5.2 Problem statement

The model. As described in §3, the *SEDA* design splits the service request processing into a set of fine grain *stages*. Each stage has a queue of *events* that are processed by threads from a thread pool dedicated to the stage (Figure 2). To summarize the terminology, a *SEDA* server receives a service request, the request triggers the internal processing of events, and once all the processing is done, the server may generate a response.

We now provide a formal description of the optimization model; see Table 1 and Figure 8 for a summary of notations. Consider a *SEDA* server with K stages, $i = 1, 2, ..., K$. A request may enter/exit the server at any of these stages. A stage may process and pass an event to the next stage, or duplicate, drop, or merge requests. The workload characteristics are modeled using the following parameters. Each stage i has arrival rate λ_i, determined by extraneous arrivals to stage i as well as arrivals to stage i from other stages. Each stage i also has its own service rate μ_i for processing events. The service rate is $\mu_i = t_i \times s_i$, where t_i is the number of threads allocated to stage i and s_i is the service rate per thread. Each thread in stage i is assumed to consume β_i proportion of a processor when actively processing events. The remaining proportion $(1 - \beta_i)$ captures the processing time that waits on synchronous calls (usually synchronous I/O). The total number of processors at the server is denoted by p.

Objective. Our framework adjusts the per-stage thread allocation to optimize an underlying performance objective, which in our case is to minimize latency - the total time that a request spends at the server.

Handling synchronous blocking calls. In addition to workloads (types of events in the stage) that fully utilize the allocated CPU we support workloads that issue synchronous blocking calls (such as synchronous I/O)[2].

[2] Although *Orleans* itself promotes using only asynchronous I/O, we have run into multiple situations where application code has to use some legacy

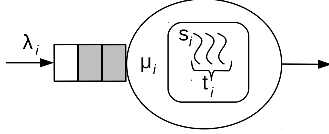


Figure 8. An illustration of a single stage in a *SEDA* server. The gray boxes represent incoming events.

Table 1. SEDA Model Notation

Notation	Description
K	Number of stages
λ_i	Arrival rate of events at each stage i
μ_i	Service rate of events at stage i
t_i	Threads allocated at stage i
s_i	Service rate per thread at stage i
β_i	fraction of processor used per thread at stage i
p	Number of processors at server

As a result, processing a single event within a stage may involve both time performing computation (denoted by x_i), and waiting for synchronous call (denoted by w_i). For those workloads allocating threads proportionally to CPU requirements only would be suboptimal. More formally, the stage's service rate per thread is $s_i = \frac{1}{x_i+w_i}$. Consider two stages i and j with $\lambda_i = \lambda_j$, $x_i = x_j$ and $w_i > w_j$. Since i waits longer for the synchronous call, we have $s_i < s_j$, and our solution will need to allocate more threads to stage i to balance out the effective CPU usage of the stages.

5.3 Latency measure and optimization

Latency measure. We first define an appropriate latency measure for each queue/stage. We choose the M/M/1 latency function given by $\frac{1}{\mu_i-\lambda_i}$, where λ_i is the arrival rate to queue i and μ_i is the service rate of the queue. We then use the weighted average of this per-queue latency across all the queues, i.e.,

$$\frac{1}{\lambda_{tot}}\sum_{i=1}^{K}\frac{\lambda_i}{\mu_i-\lambda_i}, \quad \text{where } \lambda_{tot}=\sum_{i=1}^{K}\lambda_i, \tag{1}$$

as a proxy for the end-to-end latency. This expression is known as the expected end-to-end packet delay in a *Jackson* queuing network [14], i.e., where extraneous arrivals to the network arrive according to Poisson processes, face exponentially distributed service times, and undergo independent, probabilistic routing at the queue outputs. We emphasize that we use the M/M/1 latency function as a plausible measure for representing latency, although the underlying traffic is not necessarily Poisson. Furthermore, though the end-to-end latency of a request depends on the actual topology of the network of queues, we consider a weighted sum of per-queue latency metrics for simplicity and tractability. Such approximations have been used for computer networks [14], and also for *SEDA* modeling [34]. Our evaluation (§6) shows that solving for the proxy problem results in solutions that are effective in reducing both mean and high percentile latency.

Incorporating multithreading overheads. The function (1) serves as a proxy for latency; however, it does not model overheads incurred when a large number of threads are employed. Each thread imposes additional latency due to the increased overhead of context switching. One way to model this overhead is to assume that the service rate per thread decreases with the total number of threads. This can formally be accomplished by replacing the assumption $s_it_i = \mu_i$ with a more general model, $g_i\left(t_i, \sum_j t_j\right) = \mu_i$, where $g_i(x,y)$ is a function which increases in x and decreases in y. This approach is impractical in our context because of the difficulty in modeling the functions g_i, and also since the resulting optimization problem might become complex due to the nonlinear behavior of g_i.

Instead, we incorporate multithreading overheads by adding a penalty or *regularization* term $\eta\sum_{i=1}^{K} t_i$ to the original latency cost function, where η is a suitably chosen positive constant with units of [time/threads]. The effect of the regularization is to promote solutions that favor fewer threads in total; a similar form of penalty has been used in [34]. In practice, we set η as follows – we calibrate the initial value by tuning the model on workloads with known optimal configurations. We then perform a local search around that value, and choose the one yielding the best latency results.

Optimization problem for latency minimization. Incorporating the penalty term into the end-to-end latency measure results in the following optimization problem:

$$\begin{array}{lrcll} \underset{\{t_i\}}{\text{minimize}} & \multicolumn{3}{l}{\frac{1}{\lambda_{tot}}\sum_{i=1}^{K}\frac{\lambda_i}{\mu_i-\lambda_i}+\eta\sum_{i=1}^{K}t_i} & \\ \text{subject to} & \mu_i & \geq & \lambda_i, & \forall i=1,...,K, \\ & s_it_i & = & \mu_i, & \forall i=1,...,K, \\ & \sum_i t_i\beta_i & \leq & p. & \end{array} \tag{$*$}$$

The first constraint ensures that each stage services events at least as fast as they arrive. The second constraint captures the relationship among the number of allocated threads and the associated service rates. The final constraint ensures that the available resources at the server are not exceeded.

Solution. Appealingly, $(*)$ yields a closed-form solution under plausible assumptions on the input. Formally,

library which uses synchronous I/O. Rewriting/upgrading such a library is sometimes not an option. In addition, the runtime itself may sometimes use synchronous calls. We therefore support synchronous I/O in our *SEDA* controller.

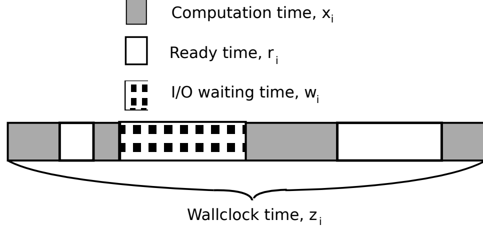


Figure 9. Time breakdown for processing one event. The width of each box represents the time elapsed while the entire width represents the wall-clock time for processing the event.

Theorem 2. *Suppose the system is feasible, i.e.,* $\sum_{i=1}^{K} \frac{\lambda_i \beta_i}{s_i} < p$*; let* $\zeta := \frac{1}{\lambda_{tot}} \left(\frac{\sum_{i=1}^{K} \beta_i \sqrt{\frac{\lambda_i}{s_i}}}{p - \sum_{i=1}^{K} \frac{\lambda_i \beta_i}{s_i}} \right)^2$*. Then if* $\eta \geq \zeta$*, the solution to (*) satisfies* $t_i = \frac{\lambda_i}{s_i} + \sqrt{\frac{\lambda_i}{\lambda_{tot} \eta s_i}}$ *for every* i*.*

The proof follows from the first-order optimality conditions of the Lagrangian of $(*)$ and is omitted for brevity. The condition $\eta \geq \zeta$ is guaranteed to be met under plausible choices of η[3]. For alternative cases where this closed form solution does not apply, the problem $(*)$ is convex and can still be solved efficiently via standard gradient methods. Observe that the number of threads is proportional to the *ratio* between the stage arrival rate λ_i and the service rate per thread s_i. Since $s_i = \frac{1}{x_i + w_i}$ the allocation takes into account both the CPU execution time and blocking time, as required.

5.4 Estimating model parameters from *Orleans* runtime measurements

The parameters required by our optimization framework of §5.3 are K, p, λ_i, s_i, and β_i. The values of K and p are directly obtained from the available information about the stages and server hardware. The other parameters would pertain to time averages, which are periodically updated using the latest measurements. The arrival rates λ_i can be deduced by averaging counts of the rates of requests and events. The parameters s_i (service rate per thread) and β_i (processor usage consumed per thread) cannot be directly measured – estimating their values requires careful measurements of events and related quantities, as we elaborate below.

To illustrate the difficulties involved in measuring s_i and β_i, let us look at what happens as a single event is processed at a given stage i. As depicted in Figure 9, processing a single event involves the time performing the computation on the processor (x_i), waiting for synchronous calls (w_i), and remaining in a ready state in an operating system scheduler until a processor is available (r_i). The wallclock time z_i is the sum $z_i = x_i + w_i + r_i$. By definition, $s_i = \frac{1}{x_i + w_i}$ and $\beta_i = \frac{x_i}{x_i + w_i}$.

[3] E.g., for high load, $[p - \sum_{i=1}^{K} \frac{\lambda_i \beta_i}{s_i}] \ll [\sum_{i=1}^{K} \beta_i \sqrt{\frac{\lambda_i}{s_i}}]$; consequently $\zeta \ll 1$ and the condition holds for most values of η.

Measuring z_i and x_i. The wallclock time z_i can be obtained by reading fine-grained system time before and after processing each event. We track the CPU time x_i by reading the cycle counter before and after each event and converting the cycle count to time based on the processor frequency. Alternatively, z_i and x_i can be measured via a fine-grained event tracing system like Event Tracing for Windows (ETW) [2], if available.

Estimating s_i and β_i. To obtain s_i and β_i we require the waiting time w_i. For a system without any synchronous calls, $w_i = 0$ for each stage. When the source code is available, w_i can often be measured directly. However, for the vast majority of production systems we cannot explicitly measure w_i because it requires knowing and measuring all the synchronous calls it uses, which may be hidden in the library. Alternatively, w_i can be obtained on platforms that provide a direct OS support for measuring I/O blocking time (such as ETW [2]). Nevertheless, we wish to develop a scheme for estimating blocking time that is more versatile and requires no direct OS support.

Our scheme proceeds as follows. Instead of estimating w_i, we estimate a related parameter, r_i, based on other available measurements. With z_i and x_i at hand, estimating r_i leads to estimations of s_i and β_i (see Fig. 9). We start by assuming that there exists a stage without any synchronous calls, as is common for a subset of stages in our workloads. For such a stage i, we know that $\beta_i = 1$ and thus $r_i = z_i - x_i$. To estimate r_i for each synchronous stage we make the assumption that $\frac{r_i}{x_i} = \frac{r_j}{x_j} \triangleq \alpha$ for all i, j[4]. We estimate α from all the stages without wait times. Formally let S_0 be a subset of stages without wait times. Then, $\alpha = \left(\sum_{i \in S_0} \frac{z_i - x_i}{x_i} \right) / |S_0|$. We can now obtain r_j for every stage j via $r_j = \alpha x_j$. With r_j at hand $s_i = \frac{1}{z_i - r_i}$ and $\beta_i = \frac{x_i}{z_i - r_i}$.

6. Evaluation Results

Testbed. The experiments are performed on a cluster of 10 servers, each with AMD 2x4 2.1 GHz Opteron processors, 16GB of RAM. All servers run 64 bit Windows Server 2008 R2 and .NET 4.5 framework. We use 15 additional frontend servers to generate client requests.

6.1 Optimized actor partitioning

We use the *Halo* Presence application (see §3) to evaluate the efficiency of *ActOp*'s actor partitioning optimization. *Halo* Presence is a representative of many applications like social networks, with extensive interaction between clients.

Workload. Our goal is to create a challenging, realistic workload with a high rate of change of the communication graph,

[4] We assume that the proportion between ready time and compute time is approximately the same for all threads. This holds for all fair OS schedulers that allocate CPU proportionally to how much a thread waits in a ready state. The longer the thread waits, the higher its priority becomes and the more CPU it gets.

and load the system to achieve a typical CPU utilization per server that we observe in production.

We target an average of 100K concurrent players in the system. Players looking for a game are placed into a game pool of 1000 idle players. 8 players are chosen at random to play a game together. The duration of a game is chosen uniformly at random between 20 and 30 minutes, reflecting typical game duration. A player may play three to five games before leaving the system. After completing a game, the player returns to the pool of idle players to find the next game. Each player stays in the system 100 minutes on average. To reach 100K concurrent players, new players arrive following a Poisson process with rate of $\frac{100K}{100}$ players per minute. This workload results in a change rate of about 1% of the communication graph (both edges and nodes) per minute.

Clients issue status requests about random players. We run the system under three different loads: 2K, 4K and 6K requests/second. The highest request rate results in 80% of CPU utilization on the servers using the baseline random partitioning. Increasing the CPU utilization beyond this threshold is avoided in production systems to accommodate unexpected load bursts. We collect a total of 7.5, 15 and 22.5 million latency measurements for each request rate respectively. We record all end-to-end latencies as observed by the clients. The experiment duration is one hour, and includes thousands of games and millions of requests.

Baseline. We partition actors randomly across servers, which is the default policy in *Orleans* (see §3). The baseline system performs no migrations during execution, whereas *ActOp* migrates actors according to the distributed graph partitioning.

Steady state measurements. We evaluate the end-to-end client latency when the system is in a steady state. We consider the system to reach a steady state when it learns the underlying communication graph and finalizes the initial round of migrations of all the actors accordingly. All the changes in the communication graph from that moment on are due to the games and players joining at random times, and therefore reflect the target graph change rate of the experiment described above. In contrast, the behavior of the system when a benchmark is started is equivalent to the case where all the players join the system all at once, which is not realistic, and therefore must be excluded from the performance measurements. As reflected in our measurements of the number of migrations in Figure 10(a), the system reaches the steady state in about 10 minutes, hence the latency of client requests reported here is recorded in the last 50 minutes of the experiment.

Algorithm convergence. Figure 10(a) shows the change in the number of remote messages over time. Within 10 minutes from the start, the proportion of remote messaging among all the inter-actor messages stabilizes at about 12%. In contrast, random partitioning results in about 90% of all actor-to-actor messages to be remote. Similarly, the actor partitioning algorithm performs more actor movements at the start and then stabilizes at about 1K actor movements per minute. Given a total of about 100K actors in a system, 1% of actors are moved each minute once the system converges, which matches the graph change rate of the experiment.

Latency under load. Figure 10(b) shows the latency CDF for the baseline and optimized actor partitioning for the 6K requests/sec load. *ActOp* reduces the 99th percentile latency by more than 3×, from 726ms to 225ms, which effectively eliminates the perception of a "sluggish" server response. To analyze this latency improvement, Figure 10(c) shows the latency CDF of actor-to-actor calls between game actors and player actors. *ActOp* results in lower latency because it reduces excess serialization, as explained in §3.

We summarize the results for different system loads in Figure 10(d). The graph shows the latency improvement calculated as $100\% \times (1 - \frac{\text{optimized}}{\text{baseline}})$. *ActOp* significantly reduces the latency for all loads, but the gains are higher as load increases. The main reason is that under higher load the queuing effect in RPC serialization stages grows, thereby amplifying the effect of co-locating communicating actors on the same server. Indeed, when we measure the average CPU utilization across the servers (10(e)), *ActOp* reduces CPU utilization per server by 25% for lower system load and by 45% for higher load, meaning that with better partitioning less CPU intensive work (serialization) is done overall.

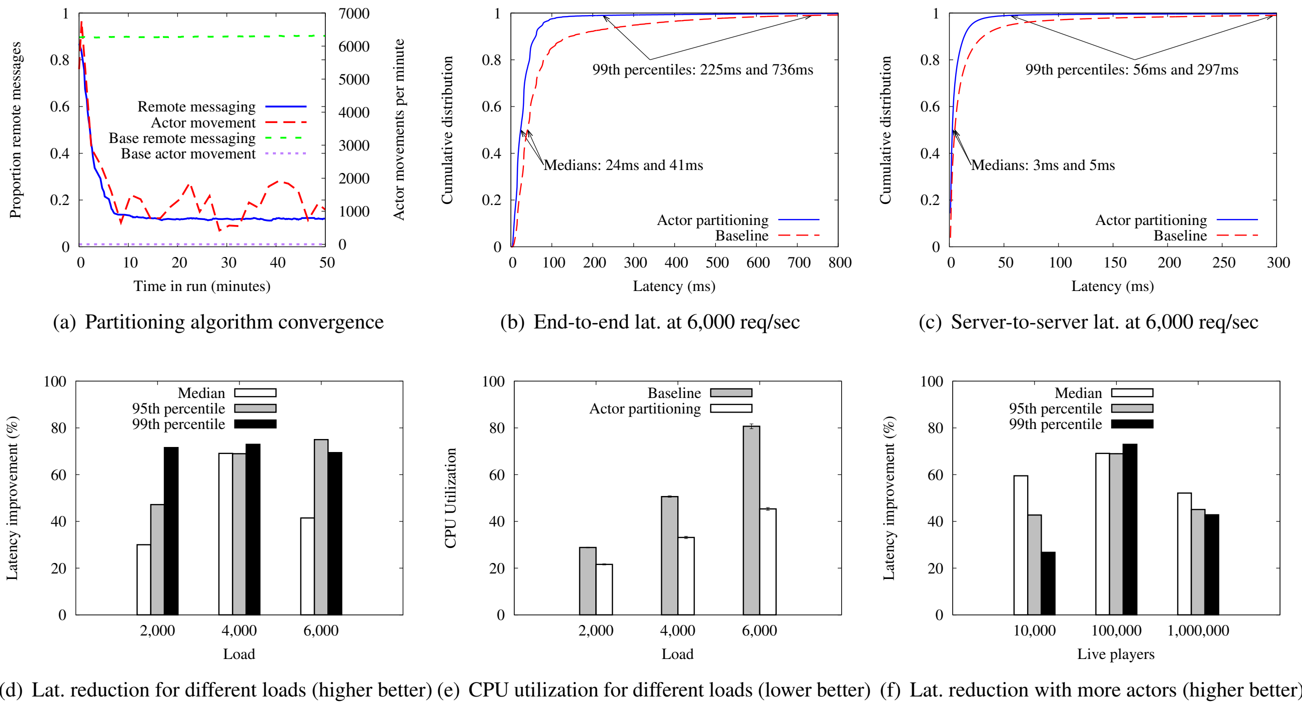
Throughput improvement. Reducing the CPU utilization for a given load enables higher throughput with the same cluster, or the same throughput on a smaller cluster. To measure the peak throughput we saturate the servers by generating requests until they start rejecting them. *ActOp* achieves the throughput of 12K requests/second, 2× higher than random partitioning which starts dropping requests at 6K requests/second while running at 80% CPU utilization.

Scaling with the number of actors. We performed experiments for 10K, 100K, and 1M live players in the system serving 4K requests/second. Additional players stress the distributed partitioning algorithm, potentially increasing its overhead and reducing its benefits. However, as Figure 10(f) shows, *ActOp* yields significant latency reduction and scales well for up to 1M actors.

6.2 Optimized thread allocation

Workload. We use the Heartbeat benchmark to evaluate the latency under *ActOp*'s optimized thread allocation. Heartbeat implements a simple monitoring service which maintains the status periodically updated by the client. This workload is similar in its call pattern to many popular services built with *Orleans*, like running statistics, aggregates or standing queries. While being a single actor application, it still requires low latency and scaling. We run the benchmark on one server and use 8 servers to generate requests. The experiment takes 25 minutes and generates 12 million latency measurements.

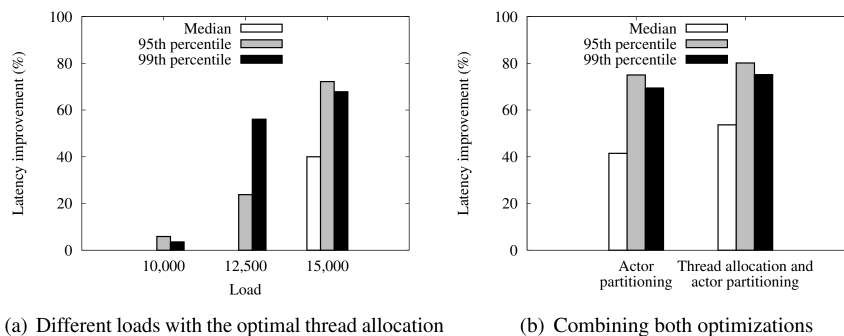
Experimental setup. *ActOp* learns the queuing model parameters, solves the optimization problem and suggests the



(a) Partitioning algorithm convergence (b) End-to-end lat. at 6,000 req/sec (c) Server-to-server lat. at 6,000 req/sec

(d) Lat. reduction for different loads (higher better) (e) CPU utilization for different loads (lower better) (f) Lat. reduction with more actors (higher better)

Figure 10. Optimizing actor partitioning for *Halo* Presence benchmark



(a) Different loads with the optimal thread allocation (b) Combining both optimizations

Figure 11. Latency improvement for different loads and different optimizations (higher is better).

optimal thread allocation for this application. We configure the system accordingly and measure the request latency. We first calibrate the model to determine the thread penalty parameter η. The calibration process involves running the Heartbeat application at 6K load and measuring its latency while varying the number of threads per stage to find the optimal configuration. We then find η which tunes the model to suggest the same configuration. After η is determined, it remains unchanged throughout the experiments. On our servers, $\eta = 100\mu sec/$thread.

The baseline thread allocation policy assigns to each stage the same number of threads, which is the number of CPU cores (eight). Intuitively, it ensures that the system is fully utilized. Other strategies assign fewer threads per stage and effectively reserve some CPU cores for specific stages, potentially leading to poor load balance across the cores.

Latency under different loads. Figure 11(a) shows the end-to-end latency with *ActOp* under different loads. We observe significant latency reduction, in particular for heavy loads. For example, the latency of the optimized configuration under the highest load of 15K requests/second is reduced by 68% in the 99th percentile, and by 58% in the median. *ActOp* allocates 2 client sender threads at each load while allocating 3 worker threads at 10K and 12.5K requests/second, and increases to 4 worker threads at 15K requests/second.

6.3 Thread allocation and actor partitioning combined

Workload and setup. We run *Halo* Presence workload with 100K players and 6K req/sec. The baseline configuration uses

default (random) actor partitioning and default (8 threads per stage) thread allocation policy. *ActOp* optimizes the system latency by both migrating actors and re-allocating threads.

Results. The optimized actor partitioning is the primary factor contributing to the latency improvement. Yet, combining both optimizations yields additional latency savings as shown in Figure 11(b): Optimizing the thread allocation provides additional reduction of 21% in the median latency, and 9% in the 99th percentile latency. In total, *ActOp* reduces the median latency by 55% and the 99th percentile by 75% over the baseline configuration. We observe that the thread allocation depends on how actors are partitioned. Applying thread allocation to random actor partitioning results in an allocation of 5 workers, 2 server senders, and 1 client sender. When actor partitioning is applied, the load on server I/O threads is reduced. Therefore, *ActOp* suggests 6 worker threads for application logic, 1 server sender, and 1 client sender. Applying both techniques together enables the best end-to-end latency.

7. Related Work

***SEDA* optimization.** Gordon [18] explores various thread allocation policies for *SEDA*, including over-allocating threads per stage and using a central thread pool shared among all the stages. Auto-tune [25] tunes the number of threads in a stage to improve the throughput. The optimization is applied to each stage individually, and it then gradually converges to an optimized thread allocation across all the stages. *ActOp* optimizes over all stages jointly, by finding the globally optimal solution which reduces the system latency. As mentioned earlier, our optimization formulation, and in particular, modeling *SEDA* as Jackson queuing network, is inspired by Welsh [34]. However, [34] uses this model to analyze the performance of *SEDA* and not directly for its optimization. The optimization proposed in [33, 34] is a greedy “local” procedure based on queue sizes. To the best of our knowledge, we are the first to tackle latency minimization via direct optimization of the underlying queuing formulation. This allows us to obtain a *global* solution (i.e., for all stages simultaneously).

Balanced graph partitioning. The balanced graph partitioning problem has been widely studied in the algorithms community for over two decades (e.g., [12, 22, 24, 29]). As mentioned earlier, the underlying techniques are centralized, requiring full graph information and typically cubic running time; this makes them prohibitively expensive for the scale we consider. Heuristics with faster running times [20, 21, 31] still require the entire graph to be in a central server, or deal with static graphs. In contrast, *ActOp* proposes a fully distributed solution targeting dynamically changing communication graphs. The work closest to ours is Ja-Be-Ja [30], which suggests a distributed balanced partitioning algorithm for large *static* graphs. As discussed in §4, [30] relies on object-to-object coordination, which does not limit the amount of per-server exchanges in a given period. As a result, the algorithm in [30] might lead to massive migration of objects (actors) in rapidly time-varying settings.

Object placement in distributed systems. SPAR [28] is a partitioning middleware for large and dynamic Online Social Network (OSN) graphs. SPAR minimizes cross server communication by dynamically creating new redundant object copies. This works well in the OSN setting, where writes are much less frequent than reads. *ActOp* uses object migrations instead of object replication, hence can cover more settings, such as object read and writes occurring at similar frequencies. Distributed Hash Tables (DHTs) [16, 19, 27] and other storage systems (e.g., [15]) also deal with object placement across servers. These systems contain no application logic and exhibit more uniform communication patterns. Accordingly, the management of data placement focuses on persistence (e.g., using additional resources for replicas), whereas cross server communication is not a significant factor.

8. Conclusion

ActOp is a data-driven optimization framework for reducing the latency of interactive applications hosted by actor systems. *ActOp* adapts to temporal changes in the actual communication graph of the application by judiciously migrating application actors across servers. Using a queuing model of a server, *ActOp* further optimizes the thread allocation in each server to match the current server load and application demands. We prototype *ActOp* in *Orleans*, and obtain substantial latency improvements for realistic workloads.

We believe that the principles and techniques we develop in *ActOp* are general and can be applied to other distributed actor systems. For example, our techniques can be applied to similar challenges of cross-server communication and suboptimal threads allocation in Akka and Erlang. However, in these systems the runtime cannot automatically and transparently move actors, making it the application’s responsibility to adjust actors placement. In contrast, the virtual nature of actors in *Orleans* enables dynamic actor migration, which allows the optimizations in *ActOp* to be integrated in the system runtime. Further, our thread allocation mechanism may be applicable to other *SEDA*-like systems that maintain multiple thread pools.

Acknowledgments

We are grateful to past and present members of Microsoft’s Project Orleans for inspiring us to work on this project, especially Phil Bernstein, Sergey Bykov, Jorgen Thelin and Alan Geller for numerous discussions and suggestions. We thank Ratul Mahajan, Jonathan Mace, Lalith Suresh, Roy Schwartz, our shepherd Nuno Preguica and the EuroSys reviewers for useful feedback which helped us improve the content and presentation of this paper. Mark Silberstein is supported by the Israel Science Foundation (grant No. 1138/14) and the Israeli Ministry of Science.

References

[1] akka. http://akka.io/.

[2] Event Tracing for Windows (ETW). https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx.

[3] Erlang. http://www.erlang.org/.

[4] Orleans. https://github.com/dotnet/orleans.

[5] Who is using Orleans. http://dotnet.github.io/orleans/Who-Is-Using-Orleans.

[6] Azure Reliable Actors. https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-actors-introduction/.

[7] Apache Camel. https://projects.apache.org/projects/camel.html.

[8] Halo using Orleans. https://gigaom.com/2014/12/15/microsoft-open-sources-cloud-framework-that-powers-halo/.

[9] Orbit. https://github.com/electronicarts/orbit.

[10] SwiftMQ. http://www.swiftmq.com/.

[11] WhatsApp Scaling. http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf.

[12] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM (JACM)*, 56(2):5, 2009.

[13] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical report, MSR Technical Report (MSR-TR-2014-41, 24). http://research.microsoft.com/apps/pubs/default.aspx?id=210931, 2014.

[14] D. P. Bertsekas and R. G. Gallager. *Data networks*. Prentice-Hall International, 2nd edition, 1992.

[15] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220, 2007.

[17] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[18] M. E. Gordon. Stage scheduling for CPU-intensive servers. *University of Cambridge, Computer Laboratory, Technical Report*, (UCAM-CL-TR-781), 2010.

[19] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Peer-to-peer systems II*, pages 98–107. Springer, 2003.

[20] G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, University of Minnesota, 1995.

[21] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[22] R. Krauthgamer, J. S. Naor, and R. Schwartz. Partitioning graphs into balanced components. In *Proc. of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 942–949, 2009.

[23] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[24] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, 46(6):787–832, 1999.

[25] Z. Li, D. Levy, S. Chen, and J. Zic. Auto-tune design and evaluation on staged event-driven architecture. In *Proc. of the 1st workshop on MOdel Driven Development for Middleware (MODDM)*, pages 1–6, 2006.

[26] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Database Theory-ICDT 2005*, pages 398–412. Springer, 2005.

[27] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at Facebook. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[28] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. *ACM SIGCOMM Computer Communication Review*, 41(4):375–386, 2011.

[29] H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proc. of the 40th ACM Symposium on Theory of Computing*, pages 255–264, 2008.

[30] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-Be-Ja: A distributed algorithm for balanced graph partitioning. In *Proc. of the 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 51–60, 2013.

[31] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proc. of the 18th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1222–1230, 2012.

[32] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, et al. Tao: How Facebook serves the social graph. In *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 791–792, 2012.

[33] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for Well-Conditioned, Scalable Internet Services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243, 2001.

[34] M. D. Welsh. *An architecture for highly concurrent, well-conditioned internet services*. PhD thesis, University of California at Berkeley, 2002.