PARTISAN: Scaling the Distributed Actor Runtime

Christopher S. Meiklejohn and Heather Miller, ***Carnegie Mellon University;***
Peter Alvaro, ***UC Santa Cruz***

https://www.usenix.org/conference/atc19/presentation/meiklejohn

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the
2019 USENIX Annual Technical Conference
is sponsored by USENIX.

PARTISAN: Scaling the Distributed Actor Runtime

Christopher S. Meiklejohn, Heather Miller
Carnegie Mellon University

Peter Alvaro
UC Santa Cruz

Abstract

We present the design of an alternative runtime system for improved scalability and reduced latency in actor applications called PARTISAN. PARTISAN provides higher scalability by allowing the application developer to specify the network overlay used at runtime without changing application semantics, thereby specializing the network communication patterns to the application. PARTISAN reduces message latency through a combination of three predominately automatic optimizations: *parallelism*, *named channels*, and *affinitized* scheduling. We implement a prototype of PARTISAN in Erlang and demonstrate that PARTISAN achieves up to an order of magnitude increase in the number of nodes the system can scale to through runtime overlay selection, up to a 38.07x increase in throughput, and up to a 13.5x reduction in latency over Distributed Erlang.

1 Introduction

Building distributed applications remains a difficult task for application developers today due to the challenges of concurrency, state management, and parallelism. One promising approach to building these types of applications is by using distributed actors; the actor-based programming paradigm is one where actors can live on different nodes and communicate transparently to actors running on other nodes. Actor-based programming is well suited to the challenges of distributed systems; actors encapsulate state, allowing controlled, serial access for state manipulation. A single machine can typically run hundreds of thousands of actors, allowing efficient use of resources per machine and thereby enabling high-scalability and high-concurrency by elastically scaling the number of machines in a cluster. Taken together with the fact that actors communicate through unidirectional asynchronous message passing with no shared memory between them, the actor-based programming paradigm is well suited to the nature of distributed systems. In addition to providing developers of distributed systems with a convenient programming model, distributed actor systems can also be efficiently implemented, which has resulted in significant adoption and large-scale success in many areas of industry.

There exist three primary industrial-grade distributed actor systems; Distributed Erlang [31], Akka [21] (for Scala) and Microsoft's Orleans [8, 10] (for C#). Distributed Erlang has been used as the underlying infrastructure for message brokers [2, 25], distributed databases [4, 6, 18], and has provided infrastructure for the chat functionality for applications like *WhatsApp*, *Call of Duty*, and *League of Legends*. [14, 15, 27] Similarly, Akka has been used by Netflix for the management of time series data [23], and Microsoft's Orleans has been used as the underlying infrastructure for Microsoft's popular online multiplayer games, *Halo* and *Gears of War* for the Xbox [24]. In all of these cases, these applications have benefited from both the state encapsulation and pervasive concurrency that actors provide and the fault isolation of actors by reducing the use of shared memory. However, these distributed actor systems are still limited in terms of both *scalability* and *latency*.

Scalability. Compared to other distributed frameworks which can support hundreds to thousands of nodes, these production-grade distributed actor systems are still limited in the number of nodes that they can support. Distributed Erlang, for instance, has not been operated on clusters larger than 200 nodes [1], whereas one of the more popular applications built on Distributed Erlang, the distributed database Riak, has been demonstrated to not scale beyond 60 nodes [15]. As we will later show, this limited scalability is related to the rigidity of the overlay network—the communication pattern between the nodes in the application—used in the runtime system. This rigidity has been the subject of previous research on alternative designs to improve the scalability of the system [11], and efforts to find a "one-size-fits-all" overlay, which can equally serve all types of distributed applications, have not been successful [28]. Thus, especially in the context of Distributed Erlang, scalability is still a major challenge.

Latency. Due to their underlying model of computation—unidirectional asynchronous message passing between ac-

tors with independent queues that are multiplexed onto a single queue between nodes—distributed actor systems frequently suffer from the problem of head-of-line blocking. For example, the distributed database Riak avoids using Distributed Erlang for background data synchronization (e.g., hinted and ownership handoff) to avoid head-of-line blocking in the read/write request path. While alleviating head-of-line blocking has been the subject of much research [12, 30] and remains a relevant problem in today's large-scale systems [9], the general solution of introducing more queues and partitioning communication across those queues does not necessarily yield better performance without a priori knowledge of the application's workload.

Application-specific information exists that can be used to reduce the effects of head-of-line blocking. Given (i) the knowledge of the identities of the actors that are sending messages, (ii) the identities of the recipients, and (iii) the knowledge that actors will process their messages sequentially, this application-specific information can be provided in the form of a small number of lightweight annotations to the runtime. These annotations can help the runtime to separate network traffic over specialized channels (e.g., cluster maintenance, high-priority application behavior, failure detection), in turn leading to the reduction of head-of-line blocking in an application-specific manner.

In this paper, we present the design of an alternative runtime system for improving the scalability and performance of distributed actor systems, along with an implementation of this runtime called PARTISAN. PARTISAN enables greater scalability by allowing the application developer to specialize the overlay network to the communication pattern required by the application at runtime without altering application semantics. PARTISAN facilities lower latency by providing the application developer with three ways to customize messaging behavior, without altering application semantics or requiring changes to application code. PARTISAN enables the application developer to (i) customize *parallelism* (for increasing the number of communication channels between nodes), (ii) utilize *named channels* (for separating different types of messages sent between actors), and (iii) *affinitize* scheduling (for partitioning traffic across communication channels depending on message source, destination and type).

We implement PARTISAN using Erlang without requiring changes to the Erlang VM, in an effort to make these scalability and latency benefits immediately available to production Erlang applications with minimal changes to application code. We provide a detailed experimental evaluation which, beyond microbenchmarks, includes a port of an existing widely-deployed Erlang distributed computing framework to take advantage of PARTISAN's optimizations. In our evaluation, we demonstrate that the use of each of these optimizations independently results in latency reduction, but the combination of these techniques yields significant reductions in latency.

The contributions of this paper are the following:

- We present the design of the PARTISAN runtime system that enables the runtime selection of overlay, enabling greater scalability by specializing the overlay to the application's communication patterns (Sections 3 & 5);
- We present a collection of predominantly automatic optimizations for latency reduction, realized in PARTISAN, that enable more efficient scheduling of messages on the network, specifically by exploiting (i) *parallelism*, (ii) *named channels*, and (iii) *affinitized* scheduling (Sections 4 & 5);
- We provide an open source implementation of PARTISAN that supports the runtime selection of overlay with implementations of four different overlay networks (Section 5);
- We port an existing widely-deployed open source distributed computing framework, Riak Core, from Distributed Erlang to PARTISAN, and provide an analysis of the process (Section 6);
- We present a detailed empirical evaluation of PARTISAN on (i) microbenchmarks, (ii) an industrial-grade actor-based distributed programming framework (Riak Core), and (iii) a research framework for distributed programming over replicated shared state (Lasp). We go on to show that PARTISAN demonstrates greater scalability (in some experiments, an order of magnitude increase in the number of nodes the system can scale to) through runtime overlay selection and lower latency (in some experiments, up to a 38.07x increase in throughput, and a 13.5x reduction in latency) through latency reduction optimizations (Section 6).

2 Background: Distributed Actors

Actors provide a simple programming model for building highly concurrent applications. Programming with actors involves two primary concepts: actors: lightweight processes that act sequentially, respond to messages from other actors, and sent messages to other actors; and asynchronous message passing: unidirectional, asynchronous messages that are sent between actors. Applications built using the actor model typically achieve their task through the cooperation of many actors sending messages to one another. No state is shared between actors: the only way for data to be shared between actors is through message passing[1]. Actors are designed to be extremely lightweight and typically implementations allow for ten to hundreds of thousands of actors per machine. As no data is shared, and actors are relatively independent with loose coupling to other actors – strictly through message passing – if a particular actor happens to fail, the fault remains isolated to that actor. Actors are not static: actors are allowed to "spawn" other actors as the system is running.

[1] Pony is a unique exception here, which uses a capability system to know when it is safe to share memory. However, this is an implementation detail as the programming model remains that of message passing.

Actors are a popular mechanism for building highly concurrent applications as they allow both users and user actions to be modeled as actors themselves. For instance, in the aforementioned *Halo* and *Call of Duty* examples, actors are use for modeling the presence service for the online functionality of the game. Therefore, a single actor, dynamically created, is used to model a connection to the service for a single user. In the Riak distributed database, an actor is spawned for every single read or write request made to the database. As the number of actors can range several orders of magnitude higher than the parallel computing capacity of a single machine, preemptive (e.g., Erlang) or cooperative scheduling (e.g., Orleans) is used for actor scheduling within the runtime.

Distributed actor systems extend the actor functionality from a single machine to a cluster of machines. Distribution adds a number of complexities to the model: (i) *failure detection*: actors may be unavailable under network partitions or crash failures of remote machines; (ii) *message omission*: messages are no longer guaranteed to arrive at a destination due to failure; (iii) *membership*: or what nodes are currently members of the cluster and how the membership overlay is organized; (iv) *binding*: the location of actors may not be known at runtime when actors are dynamically created; (v) *contention*: contention for access to network resources may slow down actors; (vi) *congestion*: and the varying location of actors results in non-uniform latency with inter-actor messaging when actors are located on different machines.

2.1 Framework Commonalities

These concerns are addressed by the contemporary industrial distributed actor systems through various mechanisms. Each of these mechanisms introduces additional network overhead that the application developer may not be aware of, contributing to reduced scalability and higher latencies.

Failure detection. Actors may become unreachable due to crash failures or network partitions. To detect failures, nodes typically send heartbeat messages to the other nodes in the cluster. When a node is suspected as failed, it's assumed that the actors that were running on that node failed.

Message omission. Distributed actor systems try to address the problem of message omission by using TCP. With a single connection, TCP ensures FIFO ordering of messages between pairs of actors and best-effort delivery using retransmission based on sequence numbers and acknowledgements.

However, as failure detection is imperfect and nodes may be disconnected and reconnected under network partitions or crash failures, message delivery is not guaranteed by the runtime system. Therefore, distributed actor systems typically require the user to program as if message omission is always a possibility. Put more generally, TCP connections are session-oriented and in these frameworks delivery guarantees do not hold across sessions.

Membership. Membership determines which nodes are part of the cluster and are available for hosting actors. Failure detection is combined with membership to determine who the active members of the cluster are at any given moment.

Binding. When sending a message from one actor to another, the location of that actor may or may not be known at a given time. Most of these systems encode a node identifier into the process identifier, or leverage a replicated, global process registry, for determining the location of an actor by a registered name instead of a process identifier.

2.2 Challenges

The problems of both network contention and network congestion remain challenges for distributed actor systems.

Network contention. All of the aforementioned actor systems support inter-machine communication through the use of a single TCP connection, therefore multiplexing actor-to-actor communication on a single channel. Not only does actor-to-actor communication (data) use this channel, but background communication from the membership and failure detection systems (control) also contribute to congestion on this link. Taken together with CPU-intensive activities that may block access to the socket (message serialization/deserialization, for example) and non-uniform distribution of message load (slow-senders vs. fast-senders), the possibility for contention increases, which in turn increases latency and reduces throughput of the system. This is further exacerbated by certain overlays; for example, the full-mesh overlay must perform failure detection from all nodes to all other nodes.

Network congestion. Network congestion, in the form of latency or congestion control, may further impact performance. Under situations where the frequency of message sends exceeds what can be transmitted over the network, causing queueing delays on these multiplexed connections between nodes, other senders on the same node may be penalized and forced to wait for other senders to transmit.

3 Overlay Networks

To address the problems that arise from a fixed overlay, PARTISAN supports the selection of overlay at runtime. PARTISAN's API exposes an overlay agnostic programming model – only asynchronous messaging and cluster membership operations – that easily allows programmers to build applications that can operate over any of the supported overlays. Selection of the overlay at runtime only affects the performance of the application, and does not change the application semantics. Selection of the overlay is done with a configuration parameter specified at runtime; therefore, changing the overlay does not require recompilation and the selection is fixed for the lifetime of the application.

PARTISAN supports four overlays and exposes an API for developers to extend the system with their own overlays: *static*, *full-mesh*, *client-server*, and *peer-to-peer*.

3.1 Static, Full-mesh, Client-server Overlays

The static, full-mesh, and client-server overlays are similar. Each overlay uses a single connection for communication between each node in the cluster. Failure detection is performed by monitoring this connection; when this connections is dropped, the node is reported as down.

With the static overlay, membership is fixed at runtime whereas with the full-mesh overlay, membership is dynamic and can be altered while the system is running. With the client-server overlay, connections are only maintained between servers and from servers to clients, similar to a traditional hub-and-spoke topology.

3.2 Peer-to-peer Overlay

The peer-to-peer overlay builds upon the HyParView [20] membership protocol and the Plumtree [19] broadcast protocol, both of which use a two-phase approach to pair an efficient dissemination protocol with a resilient repair protocol used to ensure operation during network partitions.

HyParView. HyParView is an algorithm that provides a resilient membership protocol by using partial views to provide global system connectivity in a scalable way. Using partial views ensures scalability; however, since each node only sees part of the system, it is possible that node failures break connectivity. To overcome this, HyParView uses two different partial views that are maintained with different strategies.

Plumtree. Plumtree is an algorithm that provides reliable broadcast by combining a deterministic tree-based broadcast protocol with a gossip protocol. The tree-based protocol constructs and uses a spanning tree to achieve efficient broadcast. However, it is not resilient to node failures. The gossip protocol is able to repair the tree when node failures occur.

Semantics. However, with partial views, nodes may want to message other nodes that are not directly connected. To maintain the existing semantics of existing actor systems, PARTISAN needs to support messaging between any two nodes in a cluster. To achieve this, PARTISAN's peer-to-peer membership backend uses an instance of the Plumtree protocol to compute a spanning tree rooted at each node. When sending to a node that is not directly connected, the spanning tree is used to forward the message down the leaves of the tree in a best-effort method for delivering the message to the desired node. This is similar to the approach taken by Cimbiosys [26] to prevent livelocks in their anti-entropy system.

4 Latency Reduction

In Section 2, we discussed a number of features of distributed actor systems that operate in the background to maintain cluster operation. These included *binding*, *membership*, and *failure detection*. Each of these features of actor systems can be expensive in terms of network traffic and contributes to increasing the overall message latency by delaying application-specific messaging behind cluster maintenance messaging. In addition to background traffic, it's also possible that one type of application-specific messaging may also delay different types of application-specific messaging, as in the case where a slow sender is arbitrarily delayed behind a fast sender. These are all specific cases of head-of-line blocking.

To alleviate these issues, we provide the application developer with three ways to customize messaging behavior in a distributed actor system; by (i) customizing *parallelism*, (ii) utilizing *named channels*, and (iii) *affinitized* scheduling.

4.1 Parallelism

To reduce the effects of head-of-line blocking with a single message queue, additional message queues can be introduced in an attempt to parallelize as much work as possible. We refer to this mechanism as *parallelism*. With little input from the application developer—only a specification of the number of queues to operate at each node for each destination node—the system can either use random or round-robin scheduling to assign work to queues. In most cases, the system can optimally choose this parameter based on available system resources.

4.2 Named Channels

While parallelism serves to increase the amount of work performed in parallel, background messages may be queued in front of application-specific messages, resulting in diminishing returns if this is the only technique used to reduce latency.

If we further classify these message queues as either queues for background messaging or application-specific messaging, we can be more intelligent in our scheduling. This can be achieved using *named channels*, and it is similar to Quality-of-Service (QoS) present in many modern networking systems. This mechanism only requires the application developer to annotate what type of message is being sent, and dedicated queues based on type are used for scheduling these messages. This mechanism allows the system to automatically place background messaging on a queue where it will not interfere with application-specific messaging.

4.3 Affinity

While named channels prevent background messaging from directly interfering with application-specific messaging, application-specific messaging may still suffer from interference between actors that send at different rates.

Under the assumption that multiple outgoing queues are available (parallelism), random or round-robin scheduling may still produce schedules that lead of head-of-line blocking issues. With the knowledge that actors have (i) a distinct identity (unique references which point to each actor and which can itself be exchanged), (ii) and act sequentially, we can further refine our message scheduling algorithm by selecting an outgoing message queue based on the sending actor's identity.

Feature	API	Analogous Call (Erlang)
Join node to cluster	`join(Node)`	`net_kernel:connect_node(Node)`
Remove self from the cluster	`leave()`	`net_kernel:stop()`
Return locally known peers	`members()`	`nodes()`
Forward message to registered name	`forward(Node, Name, Msg, Opts)`	`erlang:send({Name, Node}, Msg)`
Forward message to process id	`forward(Pid, Msg, Opts)`	`erlang:send(Pid, Msg)`

Table 1: PARTISAN's API

```
call(Dst, Msg, Timeout) ->
  Dst ! Msg,

  receive
    Response ->
      Response
  after
    Timeout ->
      {error, timeout}
    end
end.
```

(a) Distributed Erlang

```
call(Dst, Msg, Timeout) ->
  partisan_pluggable_peer_service_manager:forward(Dst, Msg, []),

  receive
    Response ->
      Response
  after
    Timeout ->
      {error, timeout}
  end
end.
```

(b) PARTISAN

Listing 1: Sending messages using Distributed Erlang and PARTISAN. PARTISAN's API is designed to be a drop-in replacement for Distributed Erlang.

```
%% Use `N' to partition with affinitized scheduling.
partisan_pluggable_peer_service_manager:forward(
    Dst, Msg, [{partition_key, N}])

%% Use `Channel' to partition by channel.
partisan_pluggable_peer_service_manager:forward(
    Dst, Msg, [{channel, Channel}])
```

Listing 2: Sending messages using PARTISAN. PARTISAN's API allows both affinitized scheduling and channels to be specified for a single message send.

This scheduling technique is known as *affinitized* scheduling and results in a further reduction in latency for network intensive processes by avoiding interference between different actors that send messages at different rates—for example, two actors on the same node sending at different rates to the same remote actor can be scheduled on different queues.

The application developer can take advantage of affinitized scheduling either by enabling affinitized scheduling for all messages, where a partition key is automatically derived by the system, or by annotating individual message sends with a partition key. This partition key is then concatenated with the identity of the recipient and, using a hash function, is used to select the appropriate queue. By hashing both the sender and the recipient together, the system will attempt to collocate pairwise communication between the same two actors together, providing best-effort FIFO when the system is not operating under failure.

5 PARTISAN

PARTISAN is a runtime system that enables greater scalability and reduced latency for distributed actor applications. PARTISAN improves scalability by allowing the application developer to specialize the overlay network to the application's communication patterns. PARTISAN achieves lower latency by leveraging several predominately automatic optimizations that result in the efficient scheduling of messages. PARTISAN is the first distributed actor system to expose this level of control to the application developer, improving the performance of existing actor application and enabling new types of actor applications.

5.1 Design

All three industrial-grade actor systems follow the same underlying assumptions that define the actor model. The design of PARTISAN is therefore based upon a lowest-common-denominator view of distributed actor systems. In all cases:

- actors will act sequentially, sending and receiving unidirectional, asynchronous messages;
- actors can be located on any node on the network, known only at runtime, and the system will be able to locate, though a system specific mechanism, on which machine an actor is located;
- message delivery is not guaranteed and node failures will be detected eventually.

PARTISAN follows this lowest-common-denominator view of distributed actor systems for the sake of portability of these

ideas; the same principles behind our work can be applied to realizations of PARTISAN for the other industrial-grade actor systems, such as Akka and Orleans. Applying these ideas to Akka would be straightforward, given the programming model is directly inspired by Erlang. Orleans has a slightly different programming model involving remote method invocations, but the underlying execution model is composed of unidirectional, asynchronous message sends and receives, the same as the Erlang programming model (and, extremely similar to Erlang's included RPC abstraction.)

Based on this view of actor systems, PARTISAN adds (i) the runtime selection of overlay network, and (ii) a collection of predominantly automatic latency reduction optimizations.

Latency Reduction Optimizations. PARTISAN applies the above three optimizations, *parallelism*, *named channels*, and *affinitized* scheduling (Section 4) to this lowest-common-denominator view of actor systems to achieve sometimes significant latency reduction (demonstrated in Section 6).

While some of these ideas for latency reduction have been explored in the context of networking, these optimizations are not exposed to the developer in distributed actor systems–this work is the first to do so, to the best of our knowledge.

In order to enable the application developer to directly take advantage of these optimizations when it makes sense for their application, application developers only need to specify the number of outgoing message queues (parallelism) and the types of messages that are being sent (named channels); affinitized scheduling is automatically performed by the runtime.

5.2 API

PARTISAN is designed to be a drop in replacement for Distributed Erlang, with each API command in PARTISAN providing a 1-to-1 correspondence with Distributed Erlang. The API of PARTISAN, and its corresponding calls in Distributed Erlang, is provided in Table 1 and an example of the transformation of a program from using Distributed Erlang to PARTISAN is provided in Listing 1. Performing this 1-to-1 transformation converts a Distributed Erlang application to use PARTISAN with optimizations disabled.

Like all distributed actor systems, PARTISAN's API provides both membership operations, that are used for joining/removing nodes from the cluster, and messaging operations, that are used for asynchronously sending messages. PARTISAN's programming model is both overlay-agnostic and asynchronous. Therefore, all operations return immediately and have overlay-specific behavior.

5.3 Implementation

PARTISAN is implemented as a library for Erlang and requires no modifications to the Erlang VM. This was in an effort to make PARTISAN's scalability and latency benefits immediately available to production Erlang applications with minimal changes to application code. PARTISAN is implemented in 6.7 KLOC and is available as an open source project on GitHub. This implementation of PARTISAN has several industry adopters and a growing community.

```
{partisan, [%% Enable affinity scheduling for all messages.
            {affinity, enabled},

            %% Enable parallel connections.
            {parallel, enabled},

            %% Optional: override default.
            {parallel_connections, 16},

            %% Specify available channels.
            {channels, [vnode, gossip, broadcast]},

            %% Selection of overlay.
            {membership_strategy,
             partisan_full_mesh_membership_strategy}]}.
```

Listing 3: Riak Core configuration for PARTISAN using options in Table 2 for experiments run in Section 6.2.

5.4 Configuration

Configuration options to select overlay, enable parallelism, and specify named channels are outlined in Table 2. Listing 3 demonstrates a configuration used in our Riak Core evaluation which enables parallelism, named channels, and affinitized scheduling for all messages. Users can choose to annotate message sends with a channel for targeted use of named channels and affinitized scheduling can be enabled for all messages or for an individual message; these options are demonstrated in Listing 2.

If the number of parallel connections is not specified by the user, the system will default to a reasonable value for this parameter based on the number of Erlang schedulers available. Under a default configuration of the Erlang VM, a single scheduler maps to a single vCPU. This default configuration and heuristic is discussed in detail in our experimental evaluation. (Section 6.1).

5.5 Bring Your Own Overlay

PARTISAN exposes an API for users to implement their own overlays; application developers must simply implement the `membership_strategy` interface for handling messages. PARTISAN automatically uses this membership strategy for processing incoming and outgoing messages to the system – the application developer only needs to handle internal state transitions and supplying the system with an updated list of members. PARTISAN automatically sets up required connections, serializes and deserializes messages, performs failure detection, and message forwarding. This makes it possible to implement protocols with very little code; our implementation of the full-mesh membership protocol is 152 LOC.

6 Experimental Evaluation

To evaluate PARTISAN, we designed a set of experiments to answer the following questions:

Feature	Configuration Option
Enable parallelism with default number of connections	`{parallel, enabled}`
Specify number of *N* connections to each peer	`{parallel_connections, N}`
Open *N* parallel connections for each of the named channels	`{channels, [Channel1, Channel2]}`
Enable affinitized scheduling for all messages	`{affinity, enabled}`
Specification of overlay	`{membership_strategy, MembershipStrategy}`

Table 2: PARTISAN's configuration options

- **RQ1:** What are the benefits of affinitizing actor messaging across a number of parallel TCP connections?
- **RQ2:** Can these optimizations be used on real-world applications to achieve reduction in message latencies?
- **RQ3:** Does the selection of the overlay at runtime provide better scaling properties for the application?

We begin with a set of microbenchmarks (Section 6.1), where we seek to examine the benefits of affinitizing actor communication across a number of parallel connections. We demonstrate that PARTISAN's optimizations can provide reductions in latency for workloads containing large objects, or when deployed in high latency scenarios.

Next, we examine the applicability of these optimizations on real-world applications (Section 6.2). Using a real-world distributed programming framework with an example key-value store, we show a significant reduction in latency under both high latency scenarios (datacenter-to-datacenter communication) and large object workloads through the use of a combination of optimizations: *parallelism*, *named channels*, and *affinitized* scheduling.

Finally, we explore the selection of the overlay on scaling to larger clusters (Section 6.3). We demonstrate that we can scale to order-of-magnitude larger clusters while maintaining the same application semantics by specializing the overlay at runtime to the application.

6.1 Microbenchmarks

To evaluate the optimizations in PARTISAN around latency reduction (**RQ1**), we set out to answer the following questions: (i) what is the effect of affinitizing actors; (ii) how does one know how many parallel connections to use when affinitizing actors; (iii) does affinitized parallelism benefit workloads in high latency scenarios; and (iv) does affinitized parallelism benefit workloads with large object sizes? We present a set of microbenchmarks that address each of these questions.

Experimental Setup. For the microbenchmarks, we used a single Linux virtual machine with 16 vCPUs with 64 GB of memory. On this machine, we ran two instances of the Erlang VM that communicate with one another using TCP with either a simulated RTT latency of 1ms (RTT within a single AWS availability zone) or 20ms (RTT between two availability zones in the same AWS region.) A single Linux VM is used for hosting both instances of the Erlang VM to ensure no interference from the external network and to guarantee a fixed latency during the duration of the experiment. This virtual machine is purposely kept underloaded, as to not see the effects of resource contention inside the Linux VM on latency. Each Erlang VM is configured to run 16 schedulers with kernel polling enabled.

Each of the microbenchmarks runs multiple configurations of PARTISAN under both increasing latency and payload size, with a fixed number of 10,000 messages per actor, per experiment. We consider PARTISAN with parallelism disabled, PARTISAN with parallelism, and PARTISAN with affinitized parallelism. We do not consider named channels in the microbenchmarks, as named channels and affinitized parallelism serve the same function: partitioning communication across a number of TCP connections either automatically or by using a user-specified partitioning key.

At the start of each experiment, *N* actors are spawned on each of two instances of the Erlang VM (unless otherwise specified, as in Figure 2), based on the desired concurrency level. Each actor will send a single message to an actor on the other node and wait for acknowledgement before proceeding. Experiments were run using the *full-mesh* overlay, but the optimizations are implemented for all overlays. Latency is reported as the time to send a single message from the source to the destination.

Results. We start by showing a baseline configuration of Distributed Erlang compared with PARTISAN in Figure 1. Our results show that leveraging additional connections and affinitizing communication increases performance regardless of concurrency. With 128 actors, 512KB payload, and 1ms RTT, PARTISAN with affinitized parallelism performs 1.69x better than Distributed Erlang. Considering parallelism, but without affinity, yields a 1.90x performance improvement. With a uniform workload and without the network as a bottleneck, affinitized scheduling yields a performance benefit over Distributed Erlang, but introduces a slight performance penalty when compared to purely random scheduling.

In Figure 1, the number of parallel connections is specified as 16; however, picking this number is not necessarily trivial! Figure 2 shows the effects on outliers based on the number of connections the system needs to maintain to its peers. Here, we demonstrate that 16 connections is a good choice for connections (and, the number selected as our best case in all experiments.) But why 16? 16 is selected using the heuristic that each Erlang VM is running 16 schedulers, one mapped

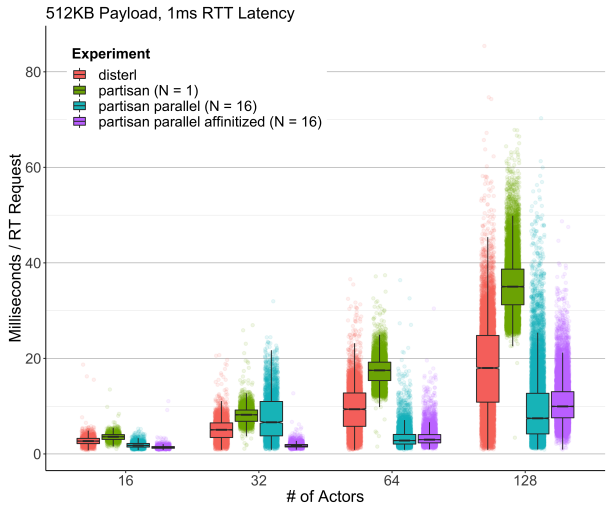

Figure 1: Performance of Distributed Erlang and PARTISAN broken out by optimization.

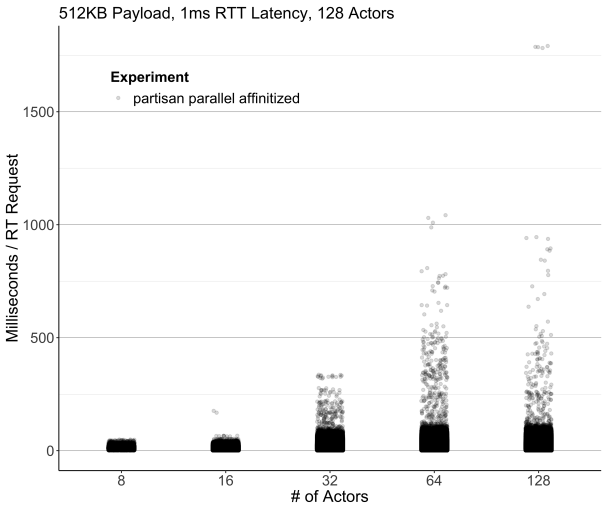

Figure 2: Effects of scaling connections with the number of actors on outliers.

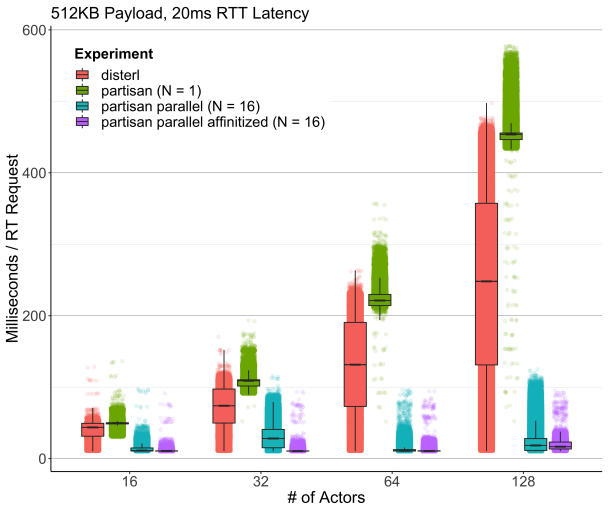

Figure 3: Performance of Distributed Erlang and PARTISAN broken out by optimization under a high latency workload: round trip time between actors is set at 20ms, object size is set at 512KB.

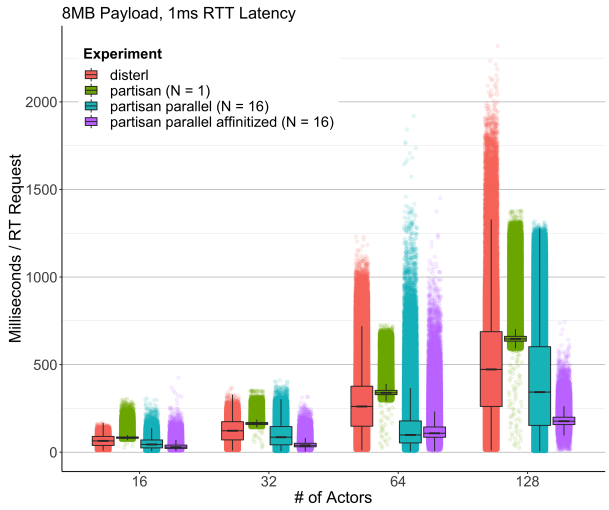

Figure 4: Performance of Distributed Erlang and PARTISAN broken out by optimization under a large payload workload: round trip time between actors is set at 1ms, object size is set at 8MB.

to a particular vCPU, and when the system needs to maintain more connections than available schedulers, context switching penalties manifest themselves as outliers (shown in Figure 2).

Focusing on these outliers, we might ask how bad does it get? With 128 actors, 512KB payload, and 1ms RTT, moving from 16 connections to 128 connections increases outliers from a max value of 176ms to 1791ms, a 10.17x increase!

In Figure 3, we turn our attention to the question of network conditions. In our first experiment (Figure 1), we chose a 1ms RTT to explore performance in a scenario where we can assume our application is running within a single AWS availability zone. But what happens if we don't have such favorable network conditions? What if our application is spread out between two AWS availability zones and suffers from RTTs closer to 20ms instead? Figure 3 shows the effects of running our earlier experiment this time with a 20ms RTT latency between actors located on different nodes. As we can see, as the latency increases, the system can take advantage of more communications channels to parallelize inter-actor communication on the network. With 128 actors, 512KB payload, and 20ms RTT, PARTISAN with parallelism performs 10.92x better than Distributed Erlang. By affinitizing parallelism, per-

formance increases to 13.50x better than Distributed Erlang.

In the Erlang community, large message sizes are not uncommon. Consider again Riak, the distributed key-value store which could contain user-stored and arbitrary-sized data. An Erlang message then could contain a user-provided piece of data megabytes in size. However, it's well-known in the Erlang community that Distributed Erlang doesn't handle large message sizes well. In fact, the Riak documentation suggests to avoid storing objects larger than 1-2MB due to the performance degradation that occurs due to Distributed Erlang [5,15]. Cognizant of this, we turn our attention to question of how large payload size affects performance in PARTISAN. Can PARTISAN overcome some of the performance issues faced by Distributed Erlang with large payloads?

Figure 4 explores the effects of increasing payload size on PARTISAN as compared to Distributed Erlang. Keeping in line with the community-observed limits of Distributed Erlang, we vary the message size from 512kb (below the 1MB performance degradation threshold) to 8MB (far above the 1MB performance degradation threshold). With 128 actors, 8MB payload, and 1ms RTT, PARTISAN with parallelism performs 1.20x better than Distributed Erlang! By affinitizing parallelism, performance increases to 2.63x.

Discussion. So far, we've seen that PARTISAN outperforms Distributed Erlang in all of our microbenchmarks. We've shown that the collection of optimizations made available to Erlang applications by PARTISAN (that is, leveraging additional connections, and affinitizing work to those connections based on the type of message and the node that the message is being sent to), can drastically improve performance by reducing latency, in some cases by over 30x.

But what does this mean practically? From these experiments, it's clear that Distributed Erlang was designed when the sort of applications being written was limited as compared to what we would like to write today; i.e., applications that send small payloads within a single data center.

As we have shown in these experiments, PARTISAN goes beyond this, and seems to be well-suited for enabling new types of applications, such as: (i) applications that operate with large data-centric workloads; (ii) applications that operate at a geo-distributed scale; (iii) the combination of both.

6.2 Evaluation: Latency Reduction in Riak

To determine the applicability of these optimizations to real-world programs (**RQ2**), we asked the following questions: (i) is it possible to modify existing application code to take advantage of the PARTISAN optimizations through the use of PARTISAN's API, and (ii) do these optimizations result in the reduction of latency for these programs?

To answer these, we ported the distributed systems framework, Riak Core, to PARTISAN and built two example applications: (i) a simple echo service – an application that's designed to only be bound by the speed of the actor receiving messages and the network itself; and (ii) a memory-based key-value store that operates using read/write quorums – more representative of a workload where more data is being transmitted and more CPU work has to occur.

6.2.1 Background: Riak Core

Riak Core is a distributed programming framework written in Erlang and based on the Amazon Dynamo [13] system that influenced the design of the distributed database Riak, Apache Cassandra, and the distributed actor framework Akka.

In Riak Core, a distributed hash table is used to partition a hash space across a cluster of nodes. These *virtual nodes*—the division of the hash space into N partitions—are claimed by a node in the cluster, and the resulting ownership is stored in a data structure known as the ring that is periodically gossiped to all nodes in the cluster. Requests for a given key are routed to a node in the cluster based on the current partitioning of virtual nodes to cluster nodes in the ring structure using consistent hashing, which minimizes the impact of reshuffling when nodes join and leave the cluster. Background processes are used for cluster maintenance; ownership handoff, (transferring virtual node ownership) metadata anti-entropy (an internal KVS for configuration metadata) and ring gossip (information about the cluster's virtual node to node mapping.)

In our experimental configuration we use 1,024 virtual nodes, the largest possible ring configuration for Riak Core. This ring size requires the largest amount of system resources – we account for this in our experiment – however, provides the most fine-grained partitioning for individual requests.

6.2.2 Modifications to Riak Core to Support PARTISAN

To perform our evaluation of PARTISAN using Riak Core, it was necessary to modify the existing application to take advantage of PARTISAN's APIs. Our changeset to Riak Core in order to use PARTISAN instead of Distributed Erlang is fairly minimal: 290 additions and 42 removals including additional logging for debugging, additional tests, and configuration.

The authors of Riak Core already realized that request traffic and background traffic could be problematic, so one mechanism inside of Riak Core—ownership handoff, responsible for moving data between virtual nodes when partitioning changes—already manages it's own set of connections. This mechanism alone contains roughly 900 LOC for connection maintenance – code that could be eliminated and replaced with calls to the PARTISAN API.

6.2.3 Echo Service

Experimental Setup. Our first application is a simple echo service, implemented on a three node Riak Core cluster. For each request, we generate a binary object, uniformly select a partition to send the request to, and wait for a reply containing the original message before issuing the next request. For each request, we draw a key from a uniform distribution over 1,024 keys – matching the ring size of the cluster – and run the key through Riak Core's consistent hashing algorithm for placement of the request. Requests originate at all of the nodes

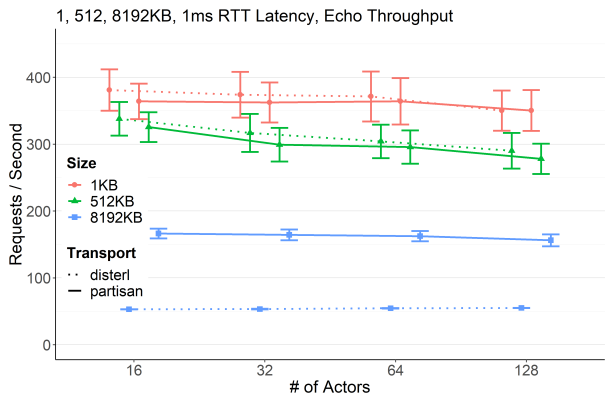

Figure 5: Performance of Distributed Erlang and PARTISAN with affinitized parallelism using the echo service / low latency workload: round trip time between actors is set at 1ms, object size varies 1, 512, and 8192KB.

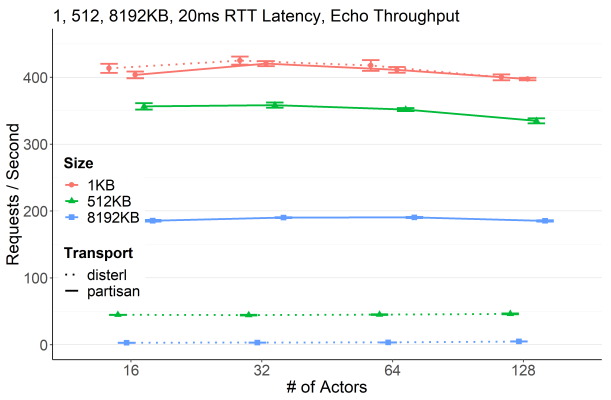

Figure 6: Performance of Distributed Erlang and PARTISAN with affinitized parallelism using the echo service / high latency workload: round trip time between actors is set at 20ms, object size varies 1, 512, and 8192KB.

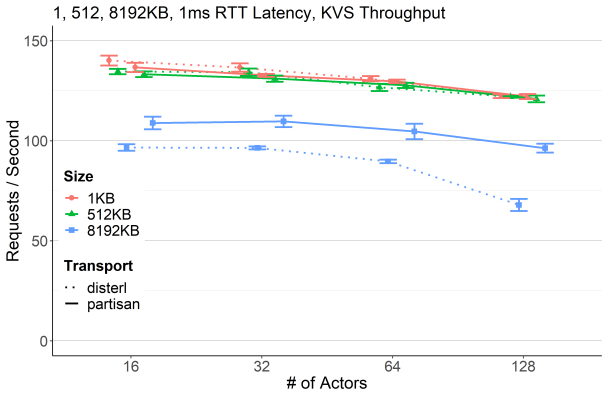

Figure 7: Performance of Distributed Erlang and PARTISAN with affinitized parallelism using the KVS / low latency workload: round trip time between actors is set at 1ms, object size varies 1, 512, and 8192KB.

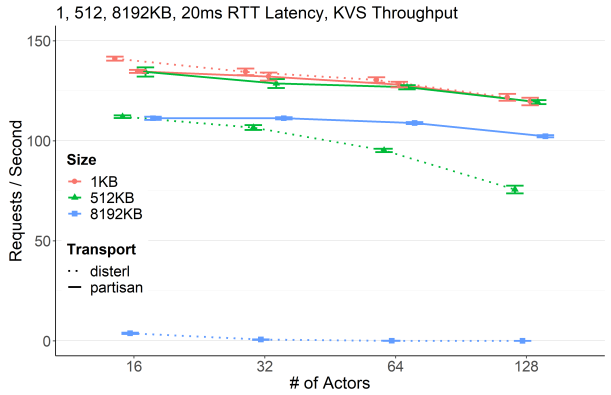

Figure 8: Performance of Distributed Erlang and PARTISAN with affinitized parallelism using the KVS / high latency workload: round trip time between actors is set at 20ms, object size varies 1, 512, and 8192KB.

in the cluster, and based on the key placement, are routed to the node responsible for handling the request. To ensure we can compare the results between runs, we wait for the cluster to stabilize before beginning the experiment.

Binary objects are generated for three payload sizes, 1KB, 512KB and 8192KB. Concurrency is increased during the test execution and parallelism is configured at 16. We test two latency configurations: 1ms, shown in Figure 5, and 20ms, shown in Figure 6. We run a fixed duration of 120 seconds.

Results. Figure 5 demonstrates that with 128 actors, 1ms RTT, and large payloads (8MB), PARTISAN is 2.84x faster than Distributed Erlang. With medium (512KB) and small payloads (1KB), PARTISAN is on par with Distributed Erlang (0.95x - 1.00x).

Figure 6 demonstrates that with 128 actors, 20ms RTT, and larger payloads (8MB), PARTISAN is 38.07x faster than Distributed Erlang (which achieves only 5 ops/second before reaching peak throughput). With medium payloads (512KB), PARTISAN is 7.25x faster than Distributed Erlang. With small payloads (1KB), PARTISAN is on par with Distributed Erlang (0.99x).

6.2.4 Key-Value Store

Experimental Setup. Our second application is a memory-based key-value store, similar to the Riak database, implemented on a three node Riak Core cluster.

Each key is hashed and mapped to a virtual node using the ring structure that is gossiped in the cluster. The virtual node that the key is hashed to, along with that virtual nodes' two clockwise neighbors on the ring, represent the three virtual nodes that contain the three replicas for the data item. Each request (either a get operation or put operation) to the key-value store uses a quorum request pattern, where requests are

made to these three replicas, and the response is returned to the user when a majority (2 out of 3) replicas reply.

This pattern involves multiple nodes in the request path, and each partition simulates a 1ms storage delay in the request path. We reuse the aforementioned benchmarking strategy: test execution is fixed at 120 seconds.

For each request, we draw a key from a normal distribution across 10,000 keys and run the key through Riak Core's consistent hashing algorithm for placement. The consistent hashing placement algorithm aims for uniform partitioning of keys across the cluster. Requests originate at all of the nodes in the cluster, and based on the key placement, are routed to the node(s) responsible for handling the request. To ensure we can compare the results between runs, we wait for the cluster to stabilize before beginning the experiment. We use a 10:1 read/write ratio for the experimental workload. Concurrency is varied in our experiments (x-axis) and parallelism is configured at 16. We test two latency configurations: 1ms, shown in Figure 7, and 20ms, shown in Figure 8.

Results. Figure 7 demonstrates that with 128 actors, 1ms RTT, and both medium (512KB) and small (1KB) payloads, PARTISAN performs on par with Distributed Erlang (0.99x-1.00x). With larger payloads (8MB), PARTISAN is 1.42x faster than Distributed Erlang.

Figure 8 demonstrates that with 128 actors, 20ms RTT, and small (1KB) payloads, PARTISAN performs on par with Distributed Erlang (0.98x). With medium payloads (512KB), PARTISAN is 1.50x faster than Distributed Erlang. With large payloads (8MB), PARTISAN far exceeds the performance of Distributed Erlang, achieving 102 ops/second; Distributed Erlang only completes 1 operation during the entire 120s execution.

6.2.5 Discussion

As we have shown in these experiments, PARTISAN is not only well-suited as a replacement for Distributed Erlang, given its similar performance under workloads that Distributed Erlang was designed for, but PARTISAN also enables new classes of applications in distributed actor frameworks. Our experiments have shown increased throughput in applications with large data-centric workloads: an example of this would be the Riak distributed database without 1MB storage limitations.

6.3 Evaluation: Improving Scalability in Lasp

In our previous experiment on latency reduction in Riak Core, we demonstrated optimizations for latency reduction in a distributed database that communicates with all of the nodes in the cluster. This is one example of an application that benefits from the *full-mesh* overlay. However, not all applications benefit from, nor require, the full-mesh model that is default case in Distributed Erlang. In this section, we address the question of whether or not an application can benefit from selection of the overlay at runtime (**RQ3**): specifically, the *client-server* and *peer-to-peer* overlays.

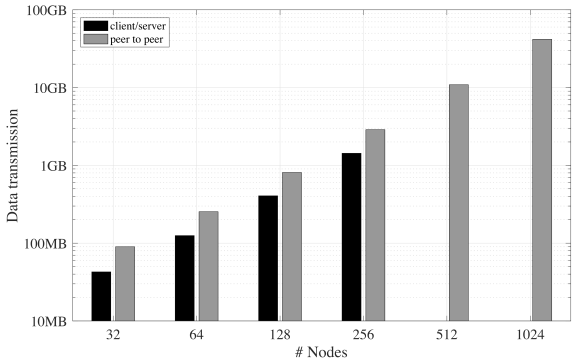

Figure 9: Comparison of data transmission for Lasp deployed on the client-server and peer-to-peer overlays for different cluster sizes (32 to 1024 nodes).

In order to understand the effect of overlay on scalability, we focus on how many nodes we can scale our application to under each overlay for an advertisement counter application implemented with Lasp.

6.3.1 Lasp

Lasp [22] is a programming framework designed for large scale coordination-free programming. Applications in Lasp are written using shared state; this shared state is stored in an underlying key-value store and is replicated between all nodes. Applications modify their own replica and propagate the effects of their changes to their peers. Lasp ensures that applications converge to the same result on every node through the use of data structures known as Conflict-Free Replicated Data Types [29], combined with monotone programming [3].

For our Lasp evaluation, the application is a simulated advertisement counter, modeled after the Rovio counter scenario for Angry Birds [22]. In this application, each client keeps a replica of a distributed counter that is incremented every time an advertisement is displayed to the user and whose state is periodically propagated to other peers in the system. When a certain number of impressions is reached, the advertisement is disabled and no longer displayed to the user.

The distributed counter used was a particular type of CRDT: a Grow-Only Counter (G-Counter). The G-Counter maps node identifiers at each of the clients to a monotonically increasing counter. Clients increment their position in the map and when merging state propagated from other nodes in the system, the pair-wise maximum is taken for each component in the map. To determine when an advertisement can be disabled, a lower bound is checked according to the sum of the components in the map: this represents a lower bound on the total number of times an advertisement has been displayed.

Experimental Setup. For this evaluation, a total of 70 m3.2xlarge Amazon EC2 instances in the same region and availability zone. Mesos [16], is used to subdivide each of these machines into smaller, fully-isolated machines. Each

container in Mesos represents a single Lasp node that communicates with other nodes in the cluster using PARTISAN.

The increment interval for each counter was fixed at 10s, and the propagation interval for the counter was fixed at 5s. The total number of impressions was configured to ensure that the experiment would run for 30 minutes under all configurations. The evaluation is performed on both the *client-server* and *peer-to-peer* overlays for different cluster sizes, ranging from 32 all the way up to 1,024 node clusters. For both overlays, the system propagates the full state of the counter to the node's peers at each propagation interval.

Note that since the Rovio advertisement counter scenario was designed for mobile applications, we do not run the full-mesh topology because it would be unrealistic. That is, in the context of mobile apps, clients would not connect to all other nodes, nor will they have knowledge of who all of the clients in the system are. Rather, either mobile apps will communicate with some number of nearby peers (peer-to-peer) or they will communicate through a server (client-server). Client-server also serves as the standard model of deploying mobile applications today. Thus, we designed our experiments to reflect this—we examine client-server and peer-to-peer overlays for this application in our experiments.

Results. Figure 9 presents the total data transmission required for the experiment to finish as we scale the size of the cluster from 32 to 1024 nodes. For smaller clusters of nodes, client-server is the more efficient overlay in terms of the amount of data that must be transmitted to finish the experiment. However, this improved efficiency comes at a cost: the client-server configuration is unable to scale beyond 256 nodes. More specifically, the experiment fails to complete because of a crash failure of the server. This crash failure occurs because of unbounded message queues: when the server is unable to process the incoming messages from the clients quickly enough, the Erlang VM allocates all available memory for storage of the message queue. This unbounded allocation results in termination of the Erlang by the Linux OOM killer once the instance runs out of available memory.

Peer-to-peer is more resilient in the face of a node failure allowing it to support larger clusters of nodes—up to 1024! However, peer-to-peer is less efficient due to this—the redundancy of communication links used by the overlay causes it to transmit more data in order to complete the experiments.

Discussion. Perhaps the most interesting takeaway from the results of this real-world large-scale experiment is that the experiment was even possible at all with Erlang. As Distributed Erlang permits one to only use a full-mesh overlay, it's possible that the previous results observed by Ericsson [1] on the maximum size of Erlang clusters–only 200 nodes–are due to this full-mesh-only restriction.

This experiment suggests that PARTISAN may enable the development of new applications with actors systems that have not been previously possible by enabling the application developer to, at runtime, change the pattern of communication between nodes, without altering application semantics. Perhaps the lack of mobile applications or even IoT applications written using distributed actor systems is a symptom of the full-mesh-only restriction.

7 Related Work

Head-of-line blocking is a well-known issue in the systems and networking community, especially in systems that use multiplexed connections. Facebook's TAO [9] relies on multiplexed connections but allows out-of-order responses to prevent head-of-line blocking issues. Riak CS [7], an S3-API compatible object storage system build on Riak, arbitrarily chunks data into 1MB segments to prevent head-of-line blocking. Geo-replicated Riak [6] contains an ad hoc implementation of node-to-node messaging to avoid Distributed Erlang at cross-region latencies. Distributed Erlang now includes a feature for arbitrarily segmenting messages into smaller chunks to reduce the impact of head-of-line blocking [17].

Ghaffari *et al.* [15] identified several factors limiting Erlang's scalability: (i) increasing payload size and (ii) head-of-line blocking with Erlang's RPC mechanism – two of the limiting factors in Riak 1.1.1's ≈ 60 node limit on scalability. Chechina *et al.* [11] proposed partitioning the graph of nodes into subgraphs and using supernodes for connecting the groups, avoiding the problems of full-mesh connectivity.

8 Conclusion

We presented PARTISAN, an alternative runtime system for improved scalability and reduced latency in actor applications. PARTISAN provides higher scalability by allowing the application developer to specify the network overlay used at runtime without changing application semantics, thereby specializing the network communication patterns to the application. PARTISAN reduces message latency through a combination of three predominately automatic optimizations: *parallelism*, *named channels*, and *affinitized* scheduling. We implemented PARTISAN in Erlang and showed that PARTISAN achieves up to an order of magnitude increase in the number of nodes the system can scale to through runtime overlay selection, up to a 38.07x increase in throughput, and up to a 13.5x reduction in latency over Distributed Erlang.

Acknowledgments

We would like to thank Scott Fritchie, Zeeshan Lakhani, Frank McSherry, Jon Meredith, Andrew Stone, Andrew Thompson, the anonymous reviewers, and our shepherd Ryan Stutsman, for their valuable feedback on this paper.

Availability

PARTISAN is available at `https://github.com/lasp-lang/partisan`. Instructions for reproducing our results are available at `https://github.com/cmeiklejohn/partisan-usenix-atc-2019`.

References

[1] Ericsson AB. Personal communication.

[2] Octavo Labs AG. Vernemq. https://vernemq.com. Accessed: 2018-02-03.

[3] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[4] Apache. Couchdb. http://couchdb.apache.org. Accessed: 2018-02-03.

[5] Basho Technologies, Inc. Developing with Riak KV. https://docs.basho.com/riak/kv/2.1.1/developing/faq/. Accessed: 2019-01-19.

[6] Basho Technologies, Inc. Riak. https://github.com/basho/riak. Accessed: 2018-02-03.

[7] Basho Technologies, Inc. Riak cs. https://github.com/basho/riak_cs. Accessed: 2018-02-03.

[8] Philip A Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, et al. Geo-distribution of actor-based services. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):107, 2017.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[10] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[11] Natalia Chechina, Phil Trinder, Amir Ghaffari, Rickard Green, Kenneth Lundin, and Robert Virding. The design of scalable distributed erlang. In *Proceedings of the Symposium on Implementation and Application of Functional Languages, Oxford, UK*, page 85, 2012.

[12] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[14] Malcolm Dowse. Erlang and First-Person Shooters. http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.pdf. Accessed: 2018-09-26.

[15] Amir Ghaffari. Investigating the scalability limits of distributed erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 43–49. ACM, 2014.

[16] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[17] Kenneth Lundin. Erlang latest news. http://erlang.org/workshop/2018/. Erlang Workshop 2018.

[18] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[19] Joao Leitao, Jose Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 301–310. IEEE, 2007.

[20] Joao Leitao, Jose Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 419–429. IEEE, 2007.

[21] Lightbend. Akka cluster documentation. https://doc.akka.io/docs/akka/2.5/index-cluster.html. Accessed: 2018-02-03.

[22] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195. ACM, 2015.

[23] Netflix. Atlas. https://github.com/Netflix/atlas. Accessed: 2018-10-01.

[24] Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 38. ACM, 2016.

[25] Pivotal. Rabbitmq. https://www.rabbitmq.com. Accessed: 2018-02-03.

[26] Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, 2009.

[27] Riot Games. Chat Service Architecture: Persistence. https://engineering.riotgames.com/news/chat-service-architecture-persistence. Accessed: 2018-09-26.

[28] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Communications of the ACM*, 53(10):72–82, 2010.

[29] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[30] Randall Stewart and Chris Metz. Sctp: new transport protocol for tcp/ip. *IEEE Internet Computing*, (6):64–69, 2001.

[31] Claes Wikström. Distributed programming in erlang. In *In PASCO'94-First International Symposium on Parallel Symbolic Computation*. Citeseer, 1994.