Making Distributed Actor Runtimes Scale

Anonymous Author(s)
Submission Id: 123-A56-BU3

Abstract

We present a collection of general techniques for latency reduction in distributed actor systems and an implementation of these ideas in the form of a programming model and distributed actor runtime called *Partisan*. *Partisan* provides higher scalability by allowing the application developer to specify the network overlay used at runtime, thereby specializing the network communication patterns to the application. *Partisan* reduces message latency through a combination of three techniques, exposed to the application developer through a programming model: *parallelism*, *named channels*, and *affinity*. We implement a prototype of *Partisan* in Erlang and demonstrate both significant latency reduction and cluster scalability using two real world applications written in Erlang.

ACM Reference Format:
Anonymous Author(s). 2018. Making Distributed Actor Runtimes Scale. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 to state manipulation. One machine can typically run on the orders of thousands or millions of actors, allowing developers to easily build highly-concurrent systems. 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 [40], Akka Cluster [26] (for the Scala programming language) and Microsoft's Orleans [10, 12] (for the C# programming language.) Distributed Erlang has been used as the underlying infrastructure for message brokers [1, 32], distributed databases [3, 8, 23], and has provided infrastructure for the chat functionality for applications like *WhatsApp*, *Call of Duty*, and *League of Legends*. [17, 20, 34] Similarly, Akka Cluster has been used by Netflix for the management of time series data [30], 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 [31]. 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, despite their success in industry, distributed actor systems are still limited in terms of both scalability and latency.

In terms of scalability, 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 [20]. These problems of scalability are related to the rigidity of the overlay network used in the runtime system and has been the subject of previous research on providing alternative designs to improve the scalability of the system; and it does not seem possible that a "one-size-fits-all" overlay exists that can equally serve all types of distributed applications [35].

In terms of latency, distributed actor systems suffer from the problem of head-of-line blocking due to actor-to-actor communication being multiplexed on a single TCP connection between nodes. While alleviating head-of-line blocking has been the subject of much research [15, 37] and remains a relevant problem in today's large-scale systems [11], 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.

Distributed actor systems can not assumed able to be automatically and transparently optimized "out of the box" for all applications that may be authored on top of them. Nor can we expect the runtime system to transparently understand a priori the behaviors of the applications that run on them. However, information exists at the level of the runtime which can be used to reduce the effects of head-of-line blocking. Given the knowledge of (a) the identities of the actors that are sending messages, (b) the identities of the recipients, and the knowledge that actors will act sequentially, lightweight annotations can provided by the application developer to

[1]Personal communication, Ericsson.

classify the type of communication between actors. By lifting the decision of how to classify network traffic over specialized channels to the level of the application developer, it is possible to reduce the effects of head-of-line blocking in an application-specific manner.

In this paper, we present a collection of general techniques for latency reduction in distributed actor systems and an implementation of these ideas in the form of a programming model and distributed actor runtime called *Partisan*. *Partisan* facilitates greater scalability in distributed actor systems by providing the application developer with 3 ways to customize messaging behavior, without altering application semantics nor requiring changes to application code. That is, *Partisan* enables the application developer to (a) customize *parallelism* (for increasing the number of communication channels between nodes), (b) utilize *named channels* (for separating different types of messages sent between actors), and (c) *affinitize* scheduling (for partitioning traffic across communication channels depending on message source, destination and type).

We implement *Partisan* using Erlang, showing the viability of shipping these changes as a library rather than as part of a Erlang's VM. 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 mechanisms. In our evaluation, we demonstrate that the use of each of these mechanisms independently results in latency reduction, but the combination of these techniques yields significant reductions in latency.

The contributions of this paper are the following:

- the design of the *Partisan* programming model and distributed runtime which supports the runtime selection of overlay;
- techniques for latency reduction (realized within *Partisan*) made available to application developers to optimize their actor programs; (a) *parallelism*, (b) *named channels*, and (c) *affinity* (Sections 4 & 5);
- an open-source implementation of *Partisan* that supports five overlays, and an open-source port of an existing widely-deployed distributed computing framework running on *Partisan* (Section 5); and
- an evaluation of *Partisan* demonstrating greater scalability through runtime overlay selection, and lower latency through latency reduction techniques, as well as several case studies demonstrating the viability of integrating these changes in today's industrial-grade distributed actor systems (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. 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.

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 or cooperative scheduling is used to map actors to the underlying hardware, in practical implementations.

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

2.1 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.

Message omission. To address the problems of message omission, distributed actor systems typically require the user to program as if message omission is always a possibility. Erlang, when operated on a single node does guarantee delivery, but suggests the programmer assume this is not true for a uniform assumption across actors on the same machine and different machines. Orleans only allows method invocations between actors – a one-way request that blocks for a one-way response with futures – and suggests the users rely on timeouts to ensure progress under message omission.

Failure detection. Actors may become unavailable for messaging due to crash failures or network partitions. Therefore, each system needs a mechanism for notifying the other nodes of the cluster of failures. Both Erlang and Orleans periodically send heartbeat messages between nodes in the cluster to try to detect when a node becomes unavailable. If a node is detected as failed, the system assumes that all actors that were running on that node have failed. While the detection mechanism is common between these two systems, Orleans and Erlang each have different mechanisms for reacting to knowledge of the failure: either by restarting actors on different nodes or taking no action at all.

Membership. Membership determines which nodes are part of the cluster and are available for hosting actors. Membership can be handled in several ways: as Orleans, using an external, strongly consistent database; as Erlang, using broadcast; or as in Akka Cluster[2], using gossip. Failure detection is combined with membership to determine who the active members of the cluster are at any given moment.

The membership overlay, full-mesh, in all three of these systems assumes that all nodes communicate with all other nodes. In Erlang's case, this results in quadratic growth in cluster metadata and exponential growth in failure detection overhead. In Akka Cluster case, failure detection is performed using adjacent nodes in the distributed hash table, resulting in reduced failure detection overhead.

Binding. When attempting to send a message to an actor, 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, like Orleans or Erlang, for determining the location of an actor by a registered name, instead of a process identifier. This is important if a logical actor might exist over several process lifecycles where it's process identifier will change, but it will answer under the same name. In Erlang, this manifests itself as failure recovery; in Orleans, this manifests itself as both failure recovery and for resource reclamation, by spinning down actors when not use and respawning them on demand.

[2]Akka Cluster's design using gossip is inherited from Riak Core's, the underlying infrastructure built for Riak in Distributed Erlang.

2.2 Challenges

However, the problems of both network contention and network congestion remain open issues.

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. Combined 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 – may lead to unnecessary contention increasing latency and reducing throughput of the system. This is further exacerbated by the fixed overlay (full-mesh) that all of the systems use by increasing background cluster maintenance traffic.

Network congestion. Furthermore, network congestion in the form of network latency may further impact performance. Under situations with increased latency, the use of a single TCP channel may arbitrarily slow down inter-node communication through TCP backpressure and congestion control mechanisms.

It is clear that all three industrial-grade actor systems exhibit a similar design and have mechanisms, albeit different, to address the same fundamental concerns. Therefore, we present our solution using a lowest-common-denominator view of distributed actor systems: 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.

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.

3.1 Overlays

Partisan supports five overlays and exposes an API for application developers to extend the system with their own overlays:

Static. *Partisan* will only connect to other nodes that have been explicitly configured at the time of node deployment time. The static overlay is designed for testing applications where the overlay should remain fixed and communication paths are identified explicitly.

Full-mesh. *Partisan* will ensure all nodes in the cluster are fully connected. Each node will connect to every other node directly, ensuring each node has full knowledge of all cluster members. This overlay is an implementation of the default configuration of Distributed Erlang. The full-mesh overlay is designed for applications where all actors may communicate with all other actors and latency should be kept to a minimum.

Client-server. *Partisan* will ensure that all nodes tagged as clients only connect to nodes tagged as server; and all nodes tagged as server nodes will connect to one another. Client-server is an implementation of the traditional overlay used by rich-web and mobile applications. The client-server overlay naturally fits applications where a lower number of server instances respond to requests from other actors that model clients; this is akin to the stateless frontend/stateful backend instances in the typical three tier architecture for elastic cloud services.

Peer-to-peer. *Partisan* will have all clients connect to at least one other client in the system and the resulting network will approximate an Erdös-Rényi [18, 19] model. The peer-to-peer overlay is designed for when the system must support a large number of clients that may need to address any actor in the system, but where low-latency is not necessarily a concern.

Publish-subscribe. *Partisan* will connect to a preconfigured AMQP [39] message broker for node-to-node messaging and dissemination of membership information. The publish-subscribe overlay is designed for environments where actors need to communicate using provided infrastructure, or where certain environmental restrictions do not allow actors to initiate connections with other actors.

3.2 Static Overlay

The static overlay assumes that nodes participating in the system will specify the nodes that they wish to connect to at deployment time: these nodes are specified in a configuration file, or in source code, and assumes a static network where nodes will not join or leave.

The static membership backend uses a single TCP connection for communication between each node in the cluster, and the failure detector reports failures when this connection drops. Static membership operates similarly to the default Distributed Erlang configuration, with the only restriction that nodes cannot be added or removed from the cluster during cluster operation.

3.3 Full-mesh Overlay

The full-mesh overlay provides the same functionality as Distributed Erlang: connections are established between all nodes in the cluster using a single TCP connection. Membership is dynamic: nodes can explicitly join or leave the cluster at any point.

3.4 Client-server Overlay

The client-server overlay assumes that each node in the system is tagged as either a client or a server. Membership is dynamic, but clients are only allowed to connect to other nodes tagged as servers and servers are optionally allowed to connect to other nodes known as server – in the case of stateless applications, like web servers this may not be required, but a partitioned database with clients of the database may need have inter-server communication for replica-to-replica communication.

The client-server overlay resembles the traditional hub-and-spoke overlay, and can be implemented by reusing the full mesh overlay, and restricting node connections between nodes based on their tags.

3.5 Peer-to-peer Overlay

The peer-to-peer overlay builds upon the HyParView [25] membership protocol and the Plumtree [24] epidemic broadcast protocol, both of which are Hybrid Gossip protocols, where a two-phase approach is used to pair an efficient dissemination protocol with a resilient repair protocol used to ensure the efficient protocol can recover from network partitions.

HyParView. HyParView is a hybrid gossip 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 failures of other nodes break connectivity or greatly increase routing length. To overcome these problems, HyParView uses two different partial views that are maintained with different strategies. The challenge is to ensure that the combination of all partial views at all nodes form a single connected component.

Plumtree. Plumtree is a hybrid gossip 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. Thus the Plumtree protocol combines the efficiency of spanning trees with the resilience of gossip.

Transitive delivery. In a HyParView cluster, nodes may want to message other nodes that are not directly connected. To maintain the existing semantics of Distributed Erlang,

Partisan needs a mechanism 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 [33] to prevent livelocks in their anti-entropy system.

3.6 Publish-subscribe Overlay

The publish-subscribe overlay builds upon the Advanced Message Queueing Protocol (AMQP) standard. AMQP is a wire-level protocol, and therefore only specifies the format messages should take. This allow *Partisan* to operate on top of arbitrary systems that support the AMQP standard, such as cloud-based offerings like Amazon's Simple Queue Service, Google's Cloud Pub/Sub and Microsoft's Azure Service Bus, and local, on-premise solutions like RabbitMQ.

Partisan's publish-subscribe overlay also only establishes outbound connections from nodes for bidirectional messaging, which makes it ideal for use in environments where outbound communication is prohibited, such as Amazon's Lambda and Google's Cloud Functions. This overlay has been used in previous work based on running Erlang in Amazon Lambda. [27]

When using the publish-subscribe overlay, a single queue is used for dissemination of membership information, that is subscribed to by all nodes participating in the system. For each Erlang node in the membership, a queue is registered for messages destined for that node; each Erlang node subscribes to its own channel.

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 information surrounding *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 3 ways to customize messaging behavior in a distributed actor system; by customizing *parallelism*, utilizing *named channels*, and *affinitized* scheduling.

To build intuition, we describe each technique in an order where latency can be successively refined.

4.1 Parallelism

Imagine as a a first attempt, the introduction of multiple outgoing message queues intended to minimize the effects of head-of-line blocking by parallelizing as much work as possible by the sender. We refer to this mechanism as *parallelism*. While this mechanism allows the system to reduce latency from processing multiple queues in parallel, if the system selects the queue under random or round-robin scheduling, background messages may still be queued in front of application-specific messages, in turn resulting in diminishing returns if this is the only technique applied to reduce latency.

4.2 Named Channels

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 allows the system to automatically place background messaging on a queue where it will not interfere with application-specific messaging. We refer to this mechanism as *named channels* and it is similar to Quality-of-Service (QoS) present in many modern networking systems. While this mechanism prevents background messaging from directly interfering with application-specific messaging, application-specific messaging may still suffer from interference between actors that have different rates of sending messaging.

4.3 Affinity

Given that actors have a (1) distinct identity (unique references which point to each actor and which can itself be exchanged), (2) and act sequentially, we can further refine scheduling by selecting a queue based on the actor's identity. This mechanism is known as *affinity* and results in further reduction in latency by reducing the effect of specific actors in application interfering with other actors sending messages at different rates, and by acknowledging that actors themselves send messages sequentially.

Normally, it is difficult to leverage affinity in the design of a general purpose programming model due to both the (a.) lack of information on the workload of the applications that will be written using it, and (b.) the fact that no single optimization suits all applications that can be written. However, distributed actor models do provide information that allow us to leverage affinitization; actors act sequentially, actors communicate pairwise, and have an identifier. In systems where several actors may share the same logical identity over time – such as in a distributed database where the actor responsible for a data item may migrate – this logical identity also provides additional information to the system.

In scheduling the communication, an outgoing message queue is selected by first selecting the logical identifier, if

present, otherwise selecting the processes identifier. That identifier, combined with the identifier of the recipient is used, along with a hashing function, to select the appropriate queue to use for message forwarding. 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.

4.4 Combining the Pieces in a Programming Model

While some of these ideas have been explored in the context of networking, these mechanisms are not exposed to the application developer in modern distributed actor systems. In order to enable the application developer to directly take advantage of these mechanisms when it makes sense for their application, we provide an API for organizing different types of message traffic sent in the cluster, and to, when necessary, pin such traffic to dedicated channels. As we will demonstrate, we can use this programmer-specified information to improve message scheduling, in turn resulting in significant latency reductions on real-world use cases.

5 *Partisan*

Partisan is a programming model and distributed runtime for Erlang. *Partisan* is the first actor runtime, to the best of our knowledge, to expose these optimization techniques directly to application developers by way of a programming model (API) and corresponding runtime.

5.1 Programming Model

Partisan's programming model resembles the typical API found in distributed actor systems. It provides two sets of operations: membership operations, that are used for joining and removing Erlang nodes from the cluster; and messaging operations, that are used for asynchronously delivering messages between nodes in the cluster.

Partisan's programming model is designed to be overlay-agnostic and fully asynchronous. Therefore, all operations in both return immediately and have overlay-specific behavior. For example, when joining a node in full-mesh mode, the node must be connected to every other node in the cluster; when joining a node in the client-server mode, if the node is a client, the node will be redirected to a server node for the connection.

Messaging is asynchronous and best-effort: in full-mesh mode, messages will be directly sent to the node; in peer-to-peer mode, a message may have to be forwarded through several nodes to reach its final destination based on what connections are available in the cluster.

The API of *Partisan* is presented in Table 1.

5.2 Implementation

Partisan is implemented as a user library for Erlang and requires no modifications to the VM. It is implemented in 6.7 KLOC and is available as an open source project on GitHub [14]. This implementation of Partisan has several industry adopters and a growing community.

Each overlay is an Erlang module that implements the `peer_service_manager` behavior. Users can implement their own overlays by providing a module that implements this behavior. Client applications interact with the Partisan system through the `peer_service` module, which exposes the API presented in Table 1. Options for forwarding messages are provided in Table 2.

Our implementation has the following optimizations:

Serialization. Serialization to Erlang's external term format occurs inside the Erlang VM and can maximize sharing of the underlying data structures before transmitting a data structure on the wire. However, since serialization is invoked outside the VM in *Partisan*, a one-time binary object is generated off-heap and immediately dereferenced once the object is transmitted. Therefore, we cannot take advantage of reusing existing, shared data structures. Using a technique from Thompson [38], recursive terms are encoded as lists and base types encoded as binaries before transmission to maximize binary reuse. This serializer is available to users who are implementing their own overlay.

Overflow of TCP LISTEN Queue. When building large clusters using the *full-mesh* overlay, the TCP LISTEN queue can overflow when other members of the cluster establish new connections. For example, a cluster of N nodes, when growing to a cluster of $N + 1$ nodes, will cause the cluster to establish N new connections to the joining node. This is exacerbated when using both the channel and parallelism features of the full-mesh overlay — a joining node will receive $N * C * P$ incoming connections at roughly the same moment, overflowing the node's LISTEN queue leading to connection timeouts. To mitigate this, when existing nodes learn about a joining node they should connect to, only establish a single connection to a joining node after every refresh interval.

Connection Cache. To avoid any unnecessary contention when sending messages, a cache (implemented as an ETS table) is used to store the list of open connections. ETS (Erlang Term Storage) tables are processes that manage shared memory storage tables in the VM that can be concurrently accessed by multiple processes for reading. This cache is available to users implementing their own overlay.

6 Experimental Evaluation

To evaluate *Partisan*, we designed a set of experiments and case studies to answer the following questions:

Table 1. *Partisan*'s API

Feature	API	Analogous Call
Join node to cluster	`join(Node)`	`net_kernel:connect(Node)`
Remove node from the cluster	`leave(Node)`	`net_kernel:stop()`
Return locally known members of the cluster	`members()`	`nodes()`
Forward message asynchronously	`forward(Node, Pid, Msg, Opts)`	`erlang:send(Pid, Msg)`

Table 2. Message forwarding options

Feature	Forwarding Option
Order messages causally based on causal label	`{causal_label, Label}`
Transitive message forwarding to destination	`{transitive, true}`
Partition key for connection affinity	`{partition_key, N}`
Send message on a particular channel	`{channel, Channel}`

- **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?

In Section 6.1, we address the question on the benefits of affinitizing actor communication across a number of parallel connections. We show a set of microbenchmarks that demonstrate that *Partisan*'s optimizations can benefit applications by providing reductions in latency for workloads containing large objects or deployed in high latency scenarios.

In Section 6.2, we address the question of the applicability of these optimizations. We demonstrate the applicability of these optimizations to a real-world distributed programming framework using an example key-value store, showing a significant reduction in latency under both high latency and large object workloads through the use of a combination of techniques: *parallelism*, *named channels*, and *affinity*.

In Section 6.3, we address the question of whether or not the selection of the overlay allows applications to scale to larger clusters. We demonstrate an application running on two different overlays and how application behavior does not change with runtime selection of the overlay, but only that the performance and data transmission does, resulting in a significant increase in cluster scalability.

6.1 Microbenchmarks

To evaluate the optimizations in *Partisan* around latency reduction (**RQ1**), we set out to answer the following questions: (a.) what is the effect of increasing parallelism on the system while scaling the number of concurrent actors?; (b.) what is the effect of affinitizing actors to a particular connection?; (c.) does affinitized parallelism benefit workloads in high latency scenarios?; and (d.) does affinitized parallelism benefit workloads with large object sizes? We present a set of microbenchmarks that address each of these questions.

For the microbenchmarks, we used a single virtual machine with 16 vCPUs with 64 GB of memory. Each of the microbenchmarks test several configurations of *Partisan* using a set of different payloads (1, 2, 4, and 8 MB), under different levels of concurrency (1, 2, 4, 8, 16, 32, and 64 processes), and a fixed number of samples (1,000 messages per experiment.)

At the start of each experiment for our microbenchmarks, N actors are spawned on each of two instances of the Erlang VM, based on the specified concurrency level. Each actor will send a single message to an actor on the other node and wait for acknowledgement before proceeding. If not specified, the latency between nodes is simulated at 1ms RTT. The system is purposely kept under load, as to not see the effects of resource contention inside the VM on latency (with the exception of Distributed Erlang, the system under test.)

Figure 1 presents our baseline. The baseline experiment compares the default configuration of Distributed Erlang with a single configuration of *Partisan* with all optimizations disabled. This demonstrates that even by implementing *Partisan* as a library, without support from the virtual machine, that *Partisan*'s performance remains competitive. However, a number of outliers do exist due to scheduling effects in the virtual machine by implementing all TCP communication as actors themselves that need to be context switched during test execution.

Introducing parallelism, and further affinitizing actors to connections, leads to an overall latency reduction as concurrency increases. Figure 2 presents results from introducing parallelism. By leveraging parallelism during the test execution, actors no longer have to block waiting for access to the single TCP connection. In this experiment, a single TCP

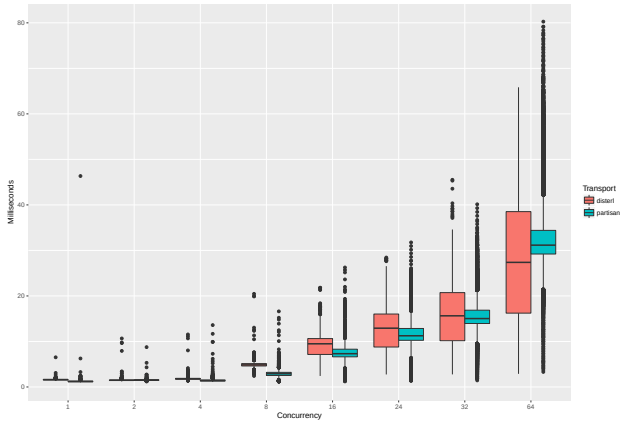

Figure 1. Baseline performance of Distributed Erlang and *Partisan*. *Partisan* remains competitive with Distributed Erlang even when implemented as a user library without Erlang VM support.

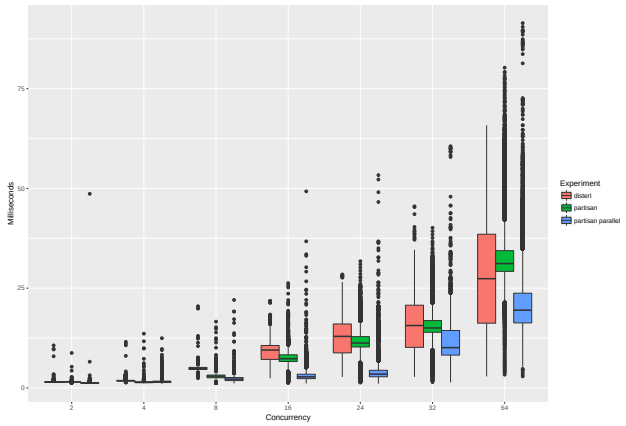

Figure 2. Performance of Distributed Erlang, *Partisan* with and without parallelism enabled. With purely random scheduling, *Partisan* suffers due to context switching inside of the Erlang VM.

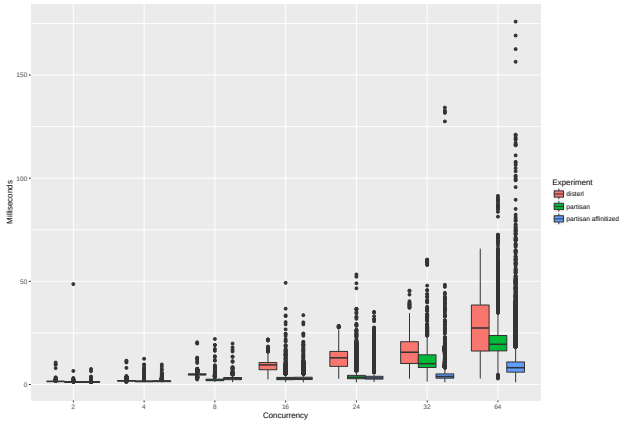

Figure 3. Performance of Distributed Erlang, *Partisan* with and without affinitized parallelism enabled. By affinitizing access to connections, performance improves as concurrency increases.

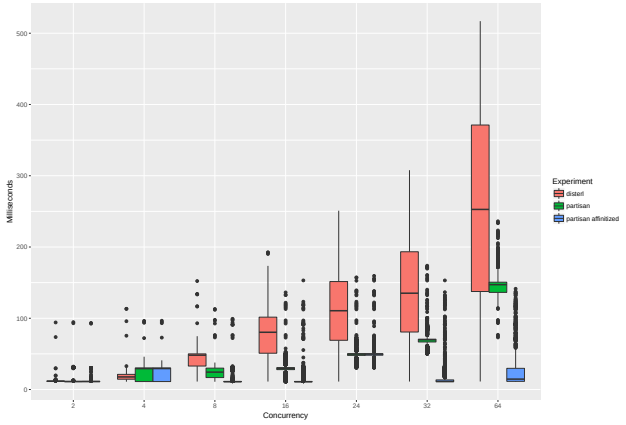

Figure 4. Performance of Distributed Erlang and *Partisan* with affinitized parallelism enabled under a high latency workload: round trip time between actors is set at 20ms, object size is set at 1MB.

connection is open for each actor (where N = Concurrency) and actors are randomly scheduled to connections using a hashing algorithm. Figure 3 demonstrates that further latency reduction can happen when actors are affinitized to a single connection.

Figure 4 demonstrates the effects of introducing more latency; in this experiment, we simulate 20ms RTT latency between actors located on different nodes. As shown, as the latency increases, the system can take advantage of more communications channels to parallelize inter-actor communication on the network. Figure 5 demonstrates a similar effect if the payload size is increased.

6.2 Case Study: Latency Reduction

To determine the applicability of these optimizations to real-world programs (**RQ2**), we asked the following questions: (a.) is it possible to modify existing application code to take advantage of the *Partisan* optimizations through the use of *Partisan*'s API, and (b.) 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: (a.) 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 (b.) a memory-based key-value store that operates using read/write quorums – more representative of

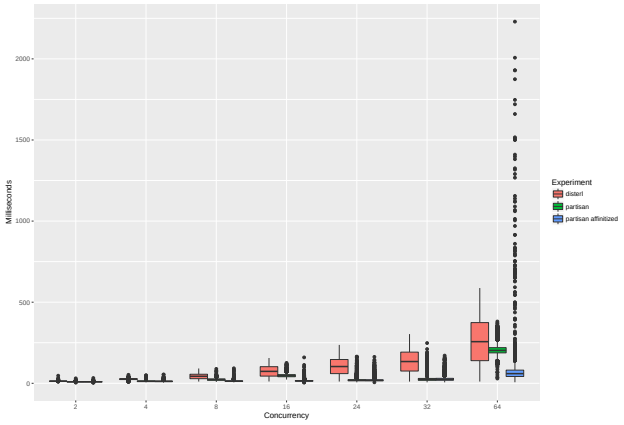

Figure 5. Performance of Distributed Erlang and *Partisan* with affinitized parallelism enabled under a large payload workload: round trip time between actors is set at 1ms, object size is set at 8MB.

a workload where more data is being transmitted and more CPU work has to occur.

6.2.1 Preliminaries

Riak Core is a distributed programming framework written in Erlang and based on the Amazon Dynamo [16] model. To perform our evaluation of *Partisan* using Riak Core, it was necessary to modify the existing application to take advantage of *Partisan*'s programming model.

Our changeset to the Riak Core application in order to use Partisan instead of Distributed Erlang is fairly minimal: 290 additions and 42 removals[3] including additional logging for debugging, adding additional tests, configuration, and required dependencies.

As part of this changeset, a new module was created that would perform all message transmission through *Partisan*. This module allowed us to, through the use of a runtime configuration parameter, alter the application to either use Distributed Erlang or *Partisan* without the modification of application code. Whenever any part of the application sends a message using this module, it specifies the type of traffic it is and provides to *Partisan* an optional parameter when forwarding the message to route the message on the appropriate channel: request traffic was additionally tagged with the optional partition key parameter, allowing the application to partition the traffic across the available parallel channels. Additionally, code was added to Riak Core to integrate its cluster join and leave operations with *Partisan*'s membership, so that joining nodes would connect to other members of the cluster through *Partisan*.

The authors of Riak Core already realized that request traffic and background traffic could be problematic, so one mechanism inside of Riak Core – virtual node 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.

In Dynamo, a distributed hash table (DHT) is used to partition a hash space across a cluster. These "virtual nodes" are claimed by a node and the resulting ownership is stored in a data structure known as the ring. Requests for a given key are routed to nodes based on the current partitioning of virtual nodes to nodes in the ring structure using consistent hashing, which minimizes the impact of reshuffling when nodes join and leave the cluster.

Background processes required for cluster maintenance, such as Riak Core's metadata anti-entropy (an internal key-value store for configuration metadata) and ring gossip (information about the cluster's virtual node to node mapping), will interleave with messages in the request path for values associated with keys, thereby creating interference, such as head-of-line blocking, if objects transmitted as part of these processes are large or that links between processes are experiencing high latency. To examine the effect of this, we ran the same unicast benchmark on a 3 node Riak Core cluster comparing Distributed Erlang and *Partisan*. *Partisan* was configured to distribute this traffic across 3 channels: request traffic, metadata anti-entropy traffic, and ring gossip.

6.2.2 Echo Service

Our first application is a simple echo service, implemented on a 3 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.

Binary objects are generated for two payload sizes, 1MB and 8MB. Concurrency is increased during the test execution and parallelism is configured at 4. We test two latency configurations: 1ms, shown in Figure 6, and 20ms, shown in Figure 7. We run a fixed duration of 120 seconds.

Affinitized parallelism provides better overall throughput by allowing the system to execute more requests in parallel. In the case of low latency and smaller object sizes, as shown in Figure 6, when the system is not performing CPU intensive work, the system can take advantage of the additional parallelism the network provides. Under high latency and larger objects, as seen in Figure 7, Distributed Erlang's performance hits a bottleneck at a certain level of concurrency where no more gains can be made.

However, affinitized parallelism does not scale linearly with the number of actors. Figure 8 demonstrates the effects of introducing too large a number of parallel connections. Due to the overhead of scheduling effects, and the additional memory and processing required to support more connections, if the number of messages increases with the concurrency level, performance degrades. In this experiment,

[3] http://github.com/lasp-lang/riak_core/pull/1

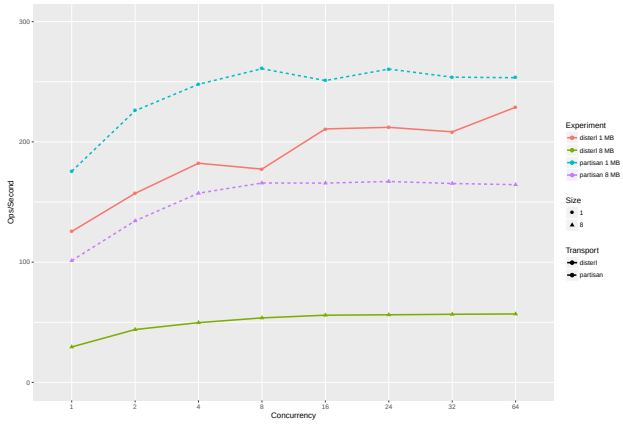

Figure 6. Echo Service: Throughput of Distributed Erlang and *Partisan* with affinitized parallelism (fixed at 4 connections) and named channels. Payload varies: 1MB and 8MB. Latency is set at 1ms RTT.

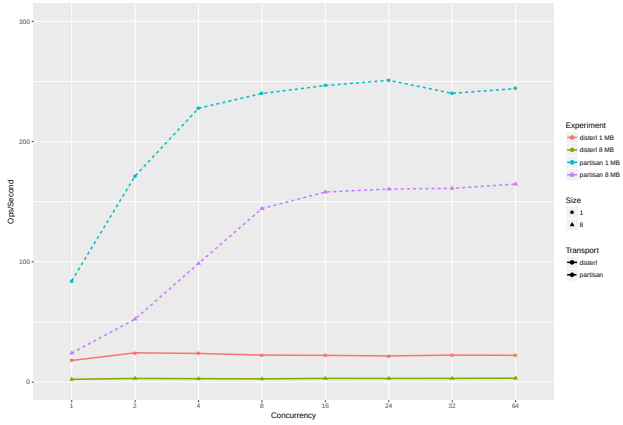

Figure 7. Echo Service: Throughput of Distributed Erlang and *Partisan* with affinitized parallelism (fixed at 4 connections) and named channels. Payload varies: 1MB and 8MB. Latency is set at 20ms RTT.

parallelism is either fixed at 4 or increased 1:1 with the number of connections.

6.2.3 Key-Value Store

Our second application is a memory-based key-value store, similar to the Riak database, implemented on a 3 node Riak Core cluster. Each request uses a quorum request pattern, where get and put requests are issued to 3 partitions, based on where the key is hashed to Riak Core's distributed hash table (along with its two clockwise neighbors), and the response is returned to the user once 2 out of the 3 partitions reply.

This pattern involves multiple nodes in the request path, and each partition simulates a 1ms storage delay[4] in the request path. Reads simulate typical key-value stores, where responses are aggregated and merged (in this case, using a Last-Writer-Wins strategy) before being returned to the user. We reuse the aforementioned benchmarking strategy: test execution is fixed at 120 seconds.

[4] 1ms is representative of a sequential seek of 1MB of data.

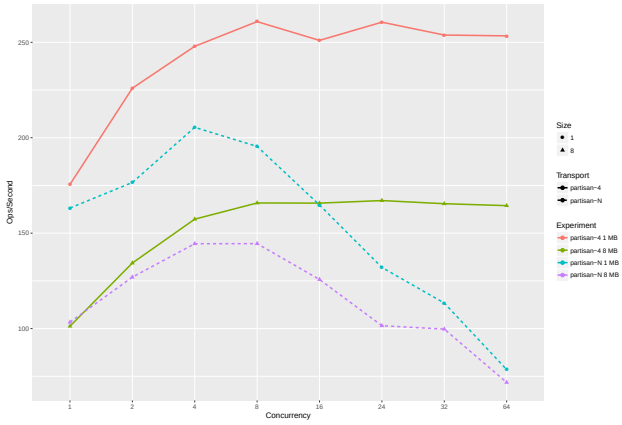

Figure 8. Throughput of *Partisan* with affinitized parallelism (fixed at 4 connections or N connections where N = Concurrency). Performance degrades when the system must maintain additional connections.

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. The workload is set at 10:1 read-to-write operations for each key for payloads of 1MB and 8MB. Concurrency is increased during the test execution and parallelism is configured at 4. We test two latency configurations: 1ms, shown in Figure 9, and 20ms, shown in Figure 10.

Under workloads that are CPU-bound, where the network is not congested nor contended for, Distributed Erlang can outperform because of (a.) the impact of scheduling overhead for *Partisan*; and (b.) its placement in the VM, which minimizes the cost of a given message send. Figure 9 demonstrates a workload where this is the case.

However, as the size of both objects and latency increases, the benefits of *Partisan* becomes apparent. Figure 10 demonstrates that *Partisan*'s optimizations allows for better performance in terms of latency and throughput, whereas Distributed Erlang begins to lose performance as concurrency increases.

6.3 Case Study: Scalability

In our case study on latency reduction, 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

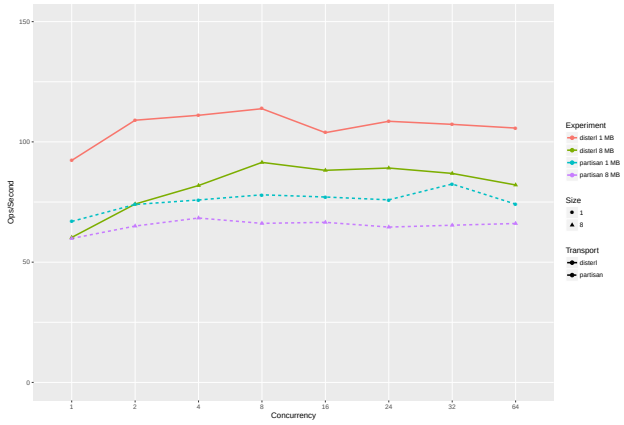

Figure 9. KVS: Throughput of Distributed Erlang and *Partisan* with affinitized parallelism (fixed at 4 connections) and named channels. Payload varies: 1MB and 8MB. Latency is set at 1ms RTT.

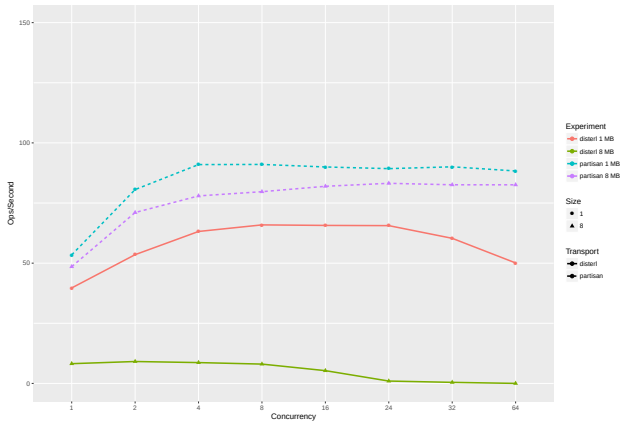

Figure 10. KVS: Throughput of Distributed Erlang and *Partisan* with affinitized parallelism (fixed at 4 connections) and named channels. Payload varies: 1MB and 8MB. Latency is set at 20ms RTT.

require, the full-mesh model. 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.

6.3.1 Lasp

Lasp [28] is a programming model 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 fully replicated between all nodes in the system. Applications always modify their own copy of the shared state, and propagate the effects of their changes to other nodes in the network. Lasp ensures that applications always converge to the same result on every node through the use of convergent data structures known

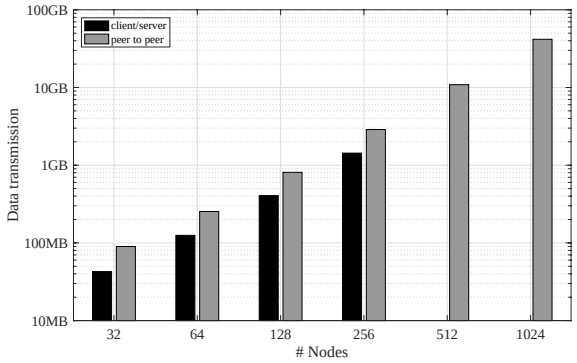

Figure 11. Comparison of data transmission for Lasp deployed on two overlay; client-server and peer-to-peer across various cluster sizes (32 to 1024 nodes).

as Conflict-Free Replicated Data Types [36], combined with monotone programming [2].

For our Lasp evaluation, the application is a simulated advertisement counter, modeled after the Rovio advertisement counter scenario for Angry Birds, where each client keeps a replica of distributed counters, issuing increment operations to each counter when an advertisement is displayed. Once a certain number of impressions are reached, the counter is disabled. The advertisement interval was fixed at 10 seconds, and the propagation interval for state was fixed at 5 seconds. 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 varying 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 objects in the local store to the node's peers at each propagation interval.

For this evaluation, a total of 70 m3.2xlarge instances in the Amazon EC2 cloud computing environment, within the same region and availability zone. Apache Mesos [21], the cluster computing framework, is used to subdivide each of these machines into smaller, fully-isolated machines using Linux's *cgroups*. Each virtual machine, representing a single Lasp node, communicated with other nodes in the cluster using *Partisan*. We refer the readers to [29] for an in-depth treatment of the large-scale Lasp evaluation. Figure 11 presents a summary of the results detailing the total data transmission required for the experiment to finish under multiple configurations. For smaller clusters of nodes, client-server is the more efficient overlay. Peer-to-peer is more resilient and can support larger clusters of nodes; however, peer-to-peer is less efficient on the network because of the redundancy of communication links used to keep the network connected.

In this experiment, data transmission is the metric used for efficiency. Data transmission is impacted by two factors:

1. **Choice of overlay.** The client-server overlay has no redundancy and uses the server as a coordination point; whereas, the peer-to-peer overlay has redundancy introduced as part of the of the overlay.
2. **Choice of data structure.** The client-server overlay has no redundancy and uses the server as a coordination point; whereas, the peer-to-peer overlay has redundancy introduced as part of the of the overlay.

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 [11], used for storage and querying of the social graph, relies on multiplexed connections but allows out-of-order responses to prevent head-of-line blocking issues. Riak CS [9], an S3-API compatible object storage system build using Distributed Erlang, arbitrarily chunks data into 1MB segments to prevent head-of-line blocking. Most recently, Distributed Erlang included a feature for arbitrarily segmenting inter-node communications into smaller chunks to reduce the impact of head-of-line blocking [22].

Some systems have been known to run different protocols on different channels. Riak Core uses an ad hoc implementation of messaging in its ownership handoff system for data migration. This implementation is roughly 900 LOC for connection lifecycle management, message serialization and transmission, and it only uses a single TCP connection for keeping handoff traffic from blocking request traffic. Zookeeper, a distributed configuration service, separates out leader election traffic from atomic broadcast traffic in the implementation. This is for historical reasons[5] since the components were developed independently and not to alleviate any specific issues related to blocking. Zookeeper therefore separates out traffic in different channels, but uses a single channel (and, single TCP connection) for all request traffic and does not attempt to parallelize requests across multiple connections to alleviate blocking problems. In fact, several open issues on the Zookeeper implementation [4–7], demonstrate issues where network partitions can prevent progress due to blocking.

Ghaffari [20] has identified several factors that limit the scalability of Distributed Erlang:

1. **Global commands.** When 0.01% of commands issued require coordination of all nodes, operations can take up to 20 seconds, limiting scalability to ≈ 60 nodes.
2. **Data size.** Increasing message sizes limit the throughput of the cluster. Ghaffari does not provide explanation for this: presumably this problem arises from head-of-line blocking and the cost of deserialization.
3. **Remote Procedure Calls (RPC).** PCs limit scalability, as each call is serialized through a single process. Head-of-line blocking, and the maximum throughput of a single process contribute to the scalability problems. The authors identified that RPCs were a limiting factor in Riak 1.1.1's ≈ 60 node limit on scalability.

Chechina et al. [13] propose that there are two challenges that must be overcome in Distributed Erlang to scale to 100s of nodes. Specifically, (i) *transitive connection sharing*, and (ii) *explicit process placement*. In the case of (i), transitive connection sharing between all nodes in the cluster requires that each instance of the Erlang VM maintains data structures quadratic in the number of nodes in the cluster. In the case of (ii), as clusters grow large enough, determining where to place becomes a challenge to ensure proper supervision, fault-tolerance, and balanced cluster performance.

The authors propose two solutions, two components of Scalable Distributed Erlang, to solve these problems:

1. **Reducing transitive connection sharing.** By partitioning the graph into subgraphs and providing full connectivity through a designated nodes in the subgraph, nodes limit the number of nodes that they have to connect to, perform failure detection on, and replicate the global process registry of. In this model, each node can become a member of multiple groups and can explicitly request a connection with another node in the system, without transitive connection sharing.
2. **Semi-explicit process placement.** When spawning a new process, per-node attributes can be used to filter the list of available nodes to choose from for hosting that process. This allows developers to target nodes by available memory, or other user-defined attributes.

8 Conclusion

We presented *Partisan* a distributed programming model and corresponding distributed actor runtime that provides better scalability for actor applications and reduced latency for actor-to-actor messaging. *Partisan* provides higher scalability by allowing the application developer to specify the network overlay used at runtime, thereby specializing the network communication patterns to the application. *Partisan* reduces message latency through a combination of three techniques, exposed to the application developer through a programming model: *parallelism*, *named channels*, and *affinity*. We have demonstrated these optimizations using microbenchmarks and two case studies using a prototype implementation of *Partisan* authored in Erlang, showing a significant improvement in scalability and latency reduction over a baseline written with Distributed Erlang. As *Partisan* assumes a least-common-denominator across all distributed actor runtimes, these optimizations can be applied to the three main distributed actor runtimes, Microsoft's Orleans, Akka Cluster, and Distributed Erlang.

[5] Personal communication, Zookeeper maintainer.

References

[1] Octavo Labs AG. [n. d.]. VerneMQ. https://vernemq.com. ([n. d.]). Accessed: 2018-02-03.

[2] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. 249–260.

[3] Apache. [n. d.]. CouchDB. http://couchdb.apache.org. ([n. d.]). Accessed: 2018-02-03.

[4] Apache Software Foundation. [n. d.]. Cannot achieve quorum when middle server (in a q of 3) is unreacable [sic]. https://issues.apache.org/jira/browse/ZOOKEEPER-2386. ([n. d.]). Accessed: 2018-05-15.

[5] Apache Software Foundation. [n. d.]. fast leader election keeps failing. https://issues.apache.org/jira/browse/ZOOKEEPER-2164. ([n. d.]). Accessed: 2018-05-15.

[6] Apache Software Foundation. [n. d.]. FLE implementation should be improved to use non-blocking sockets. https://issues.apache.org/jira/browse/ZOOKEEPER-900. ([n. d.]). Accessed: 2018-05-15.

[7] Apache Software Foundation. [n. d.]. Server fails to join quorum when a peer is unreachable (5 ZK server setup). https://issues.apache.org/jira/browse/ZOOKEEPER-1678. ([n. d.]). Accessed: 2018-05-15.

[8] Basho Technologies, Inc. [n. d.]. Riak. https://github.com/basho/riak. ([n. d.]). Accessed: 2018-02-03.

[9] Basho Technologies, Inc. [n. d.]. Riak CS. https://github.com/basho/riak_cs. ([n. d.]). Accessed: 2018-02-03.

[10] 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. 2017. Geo-distribution of actor-based services. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 107.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. 2013. TAO: Facebook's Distributed Data Store for the Social Graph.. In *USENIX Annual Technical Conference*. 49–60.

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

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

[14] Christopher S. Meiklejohn. [n. d.]. Partisan source code repository. https://github.com/lasp-lang/partisan. ([n. d.]). Accessed: 2018-02-03.

[15] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

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

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

[18] Paul Erdos and Alfréd Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci* 5, 1 (1960), 17–60.

[19] Paul Erdős and Alfréd Rényi. 1964. On the strength of connectedness of a random graph. *Acta Mathematica Academiae Scientiarum Hungarica* 12, 1-2 (1964), 261–267.

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

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.

[22] Kenneth Lundin. [n. d.]. Erlang Latest News. http://erlang.org/workshop/2018/. ([n. d.]). Erlang Workshop 2018.

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

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

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

[26] Lightbend. [n. d.]. Akka Cluster Documentation. https://doc.akka.io/docs/akka/2.5/index-cluster.html. ([n. d.]). Accessed: 2018-02-03.

[27] Christopher Meiklejohn, Zeeshan Lakhani, and Heather Miller. 2018. Towards a Solution to the Red Wedding Problem. (2018).

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

[29] Christopher S. Meiklejohn, Vitor Enes, Junghun Yoo, Carlos Baquero, Peter Van Roy, and Annette Bieniusa. 2017. Practical Evaluation of the Lasp Programming Model at Large Scale: An Experience Report. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP '17)*. ACM, New York, NY, USA, 109–114. https://doi.org/10.1145/3131851.3131862

[30] Netflix. [n. d.]. Atlas. https://github.com/Netflix/atlas. ([n. d.]). Accessed: 2018-10-01.

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

[32] Pivotal. [n. d.]. RabbitMQ. https://www.rabbitmq.com. ([n. d.]). Accessed: 2018-02-03.

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

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

[35] Rodrigo Rodrigues and Peter Druschel. 2010. Peer-to-peer systems. *Commun. ACM* 53, 10 (2010), 72–82.

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

[37] Randall Stewart and Chris Metz. 2001. SCTP: new transport protocol for TCP/IP. *IEEE Internet Computing* 6 (2001), 64–69.

[38] Andrew Thompson. [n. d.]. Chasing Distributed Erlang. http://vagabond.github.io/programming/2015/03/31/chasing-distributed-erlang. ([n. d.]). Accessed: 2018-02-03.

[39] Steve Vinoski. 2006. Advanced message queuing protocol. *IEEE Internet Computing* 10, 6 (2006).

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