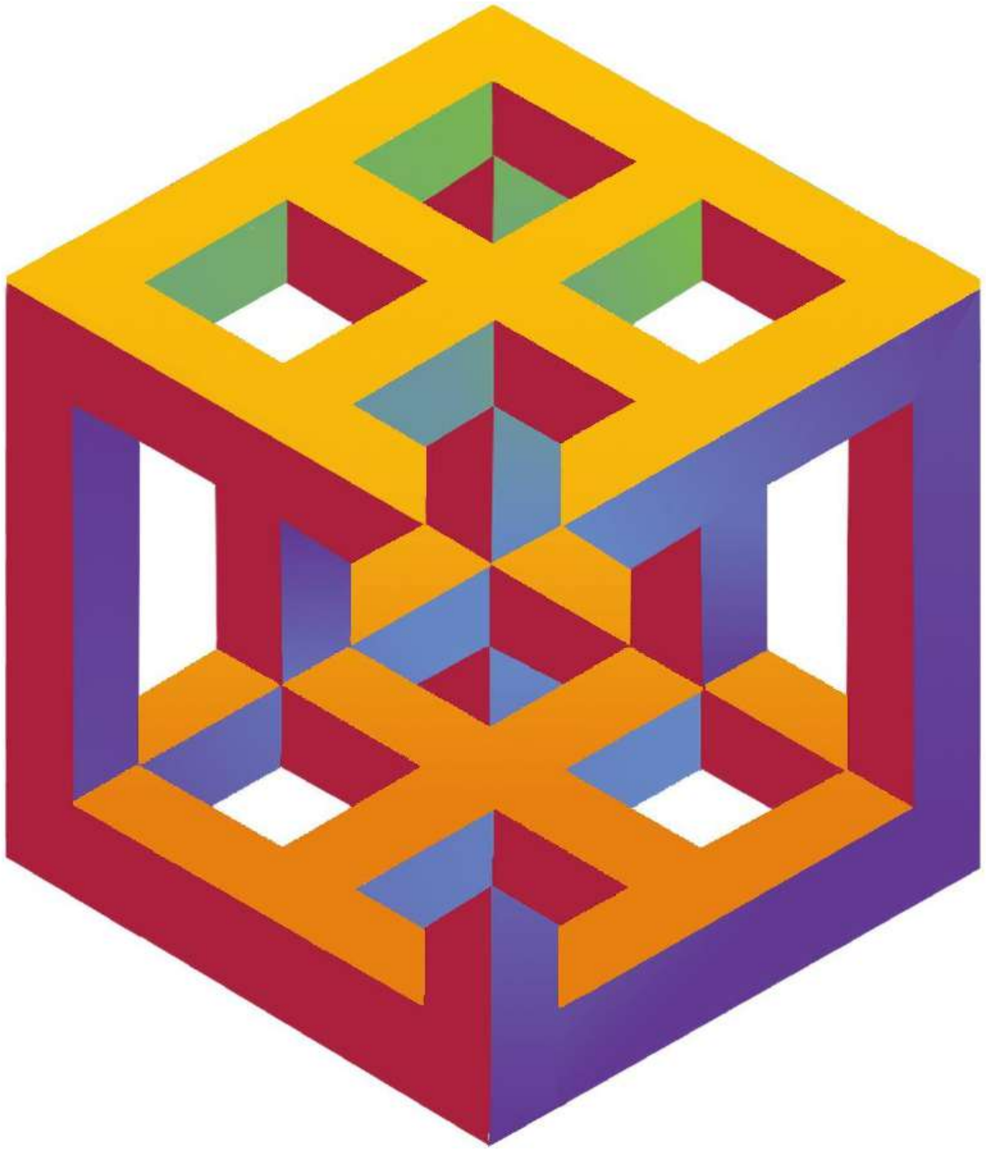
TESTABLE HIGH-PERFORMANCE LARGE-SCALE DISTRIBUTED ERLANG

Christopher S. Meiklejohn
Heather C. Miller
Peter Alvaro

Carnegie Mellon University
Baskin Engineering UC SANTA CRUZ
isr institute for SOFTWARE RESEARCH



MOTIVATION

What are actors used for and what are the problems with actors?

MOTIVATION

Distributed systems programming is still very hard:

- How to manage state?
- How do we manage concurrency?

Modern actor systems are still limited in terms of both scalability and latency!

- Encapsulation for state
- Pervasive concurrency – thousands of actors working together
- Asynchronous messaging – no shared memory between actors

Demonstrated success:

- Erlang: Call of Duty, League of Legends, WhatsApp
- Orleans: Halo, Gears of War

ACTOR EXAMPLE: DISTRIBUTED ERLANG

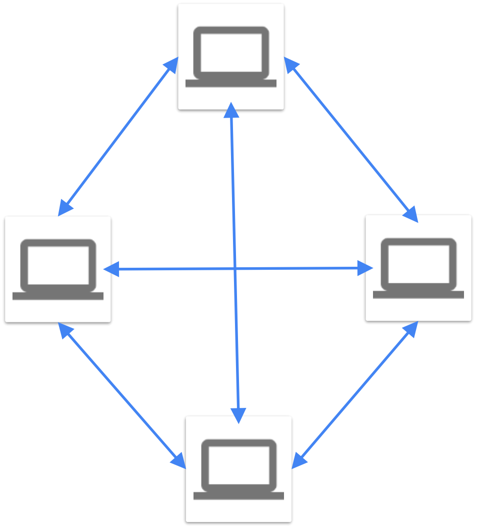
```erlang
call(Dst, Msg, Timeout) ->
  Dst ! Msg,

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

Pid = spawn(fun() -> call(OtherPid, Message, 1000) end).
```

Send a message to destination process identifier.

Wait for a response until timeout and return either the response or error.

Spawn actors running functions that message other actors.

DISTRIBUTED ACTORS: TODAY'S DESIGN



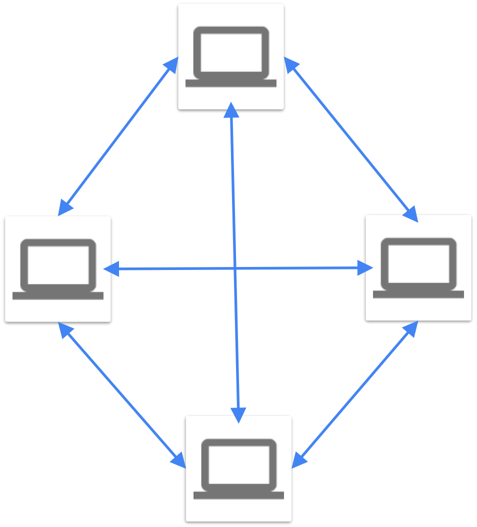
All nodes communicate with all other nodes.

- Nodes run actors that can communicate with other actors
- Transparent messaging

Nodes maintain open TCP connections.

- Heartbeat other nodes to detect failure
- Actors considered failure under partition or node failure

DISTRIBUTED ACTORS: TODAY'S DRAWBACKS



Scalability

- All-to-all communication is expensive and prohibitive
- Nodes need to know about all other nodes

Latency

- Multiplexed TCP connection is a bottleneck
- Many actors reduced to a single connection's speed
- Congestion:
  - network latency, queueing delay
- Contention:
  - competing for shared resources, slow-sender vs. fast-sender

PARTISAN

Improving the scalability of distributed actor systems.

PARTISAN

Design of an alternative runtime system for distributed actor systems

- Design and prototype implementation in Erlang

Runtime selection of communications overlay network

- Specialize overlay selection to communications pattern of application
- No modification to application code

Provides reduced latency and increased scalability

- Enable parallelism on the network
- Schedule messages efficiently on the network

PARTISAN: API

Simple transformation for existing applications to use Partisan.

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

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

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

  receive
    Response ->
```

1-to-1 correspondence in API

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)`

CAVEAT EMPTOR

References

- Unique references generated by BEAM, guaranteed globally unique
- Not serializable presently because deserialization tied to Distributed Erlang
- **Lots of platform-agnostic alternatives**: Snowflake IDs, Logical Clock derivatives (HLC, etc.)

Closures

- Subject of **my Ph.D. advisor's thesis**
- Serialization tied to Distributed Erlang
- When are these safe to capture?
- No support for sending closures at the moment

IMPROVING SCALABILITY

There's no "one-size-fits-all" overlay for distributed applications.

OVERLAY SELECTION

No "one-size-fits-all" topology

- Rigidity of the full-mesh overlay assumes one application design
- Not necessarily true for modern applications (mobile, IoT)

Selection of overlay at runtime

- Select the runtime based on the communication pattern
- Full-mesh, **Client-server**, **Peer-to-peer**, Publish-subscribe.

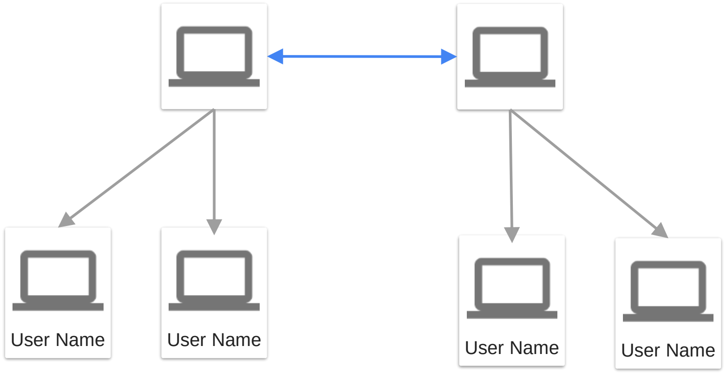
Tradeoffs

- Redundant, large-scale overlays more expensive in transmission but support more clients

```
{partisan, [%% Enable affinity scheduling.
             {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} ]}.
```

Select the overlay network desired.

CLIENT-SERVER OVERLAY



Client nodes communicate with server nodes.

Server nodes communicate with one another.

Point-to-point messaging through the server.

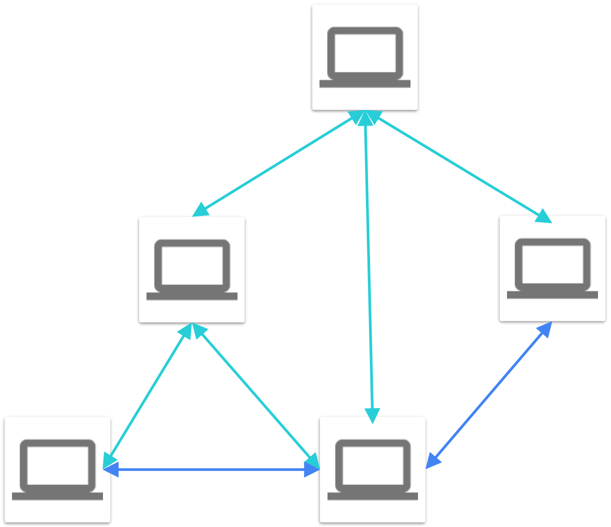
- Server routes messages on clients behalf

Nodes maintain open TCP connections.

- Considered “failed” when connection is dropped.

Typical communication pattern in **mobile** and **web** applications today.

PEER-TO-PEER OVERLAY



Supports large-scale networks (10,000+ nodes)

- Built on existing protocols: HyParView, Plumtree, Cimbiosys

Nodes maintain partial views of the network

- Active views form connected graph
- Passive views for backup links used to repair graph under failure

Nodes maintain open TCP connections.

- Considered "failed" when connection is dropped.
- Some links to passive nodes kept open for "fast" replacement of failed active nodes

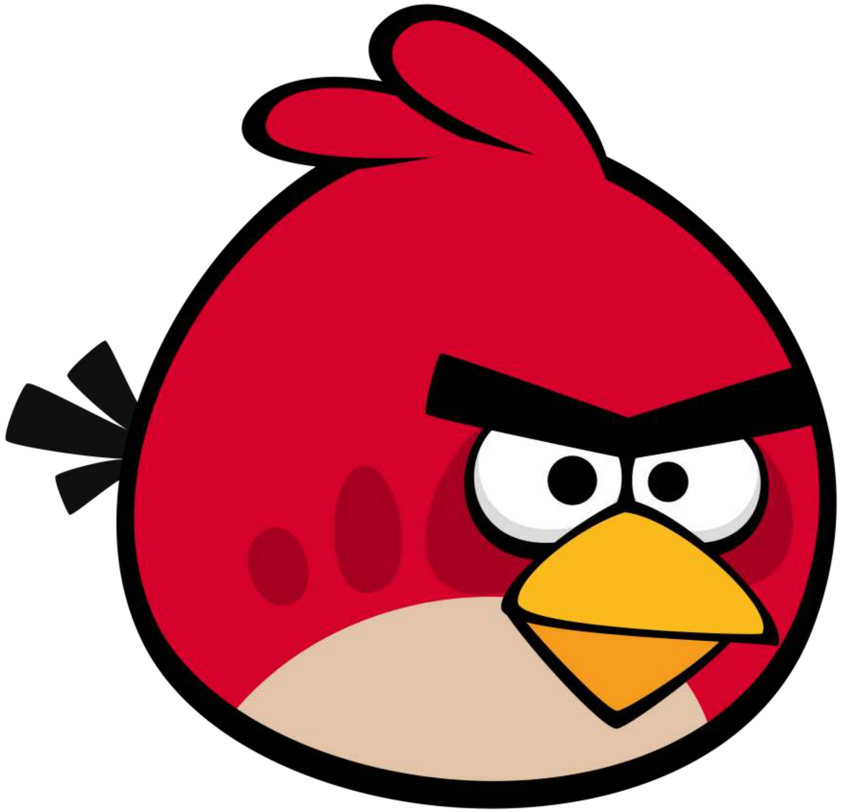
Point-to-point messaging for connected nodes.

- Spanning tree lazily computed and used for routing messages transitively to the final recipient

EVALUATING SCALABILITY

There's no "one-size-fits-all" topology for distributed applications.

ROVIO / ANGRY BIRDS

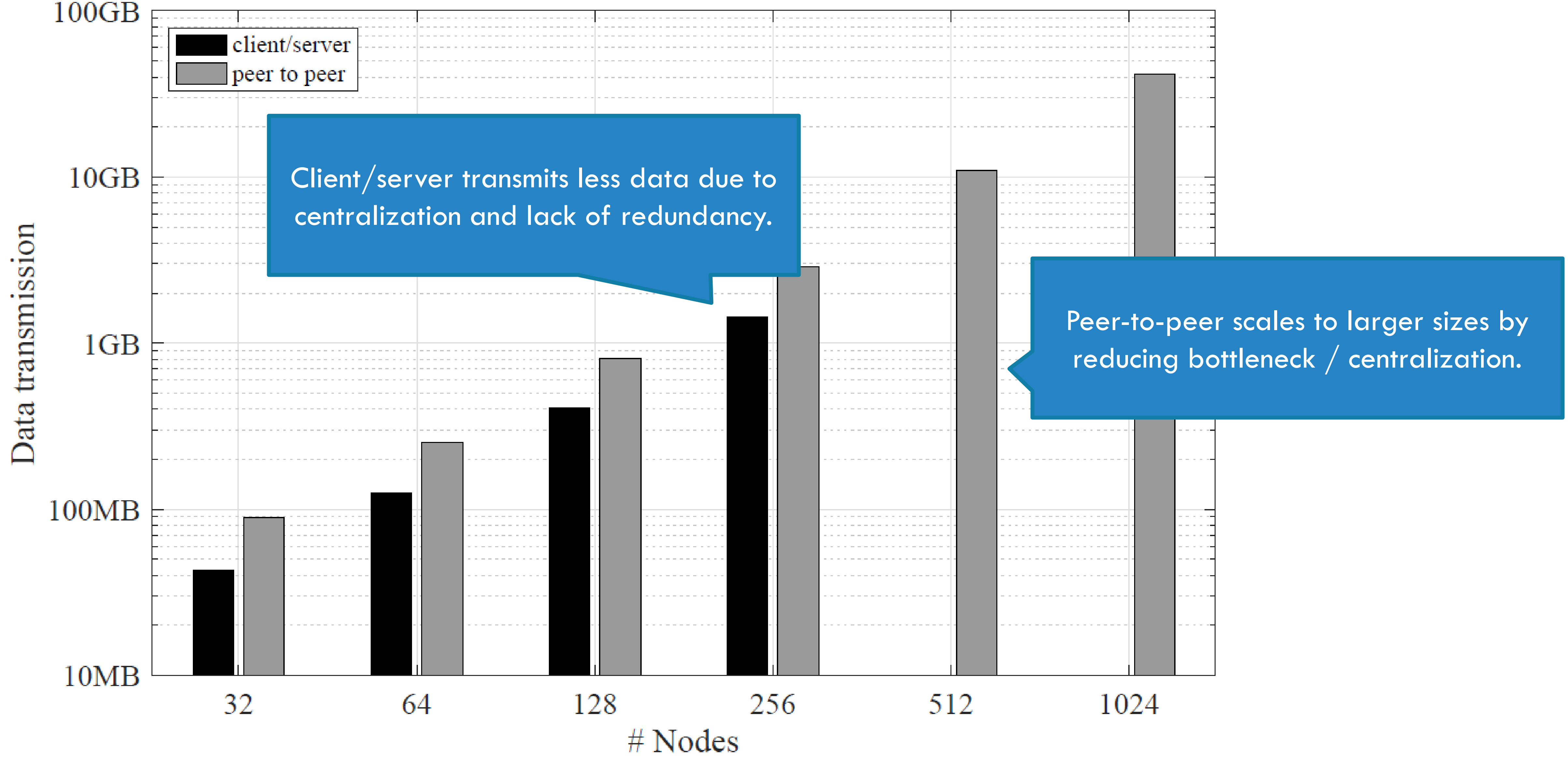


Advertisement counter (SyncFree, EU-FP7)

- Each mobile device keeps track of a counter of times displayed
- Modeled as a convergent data structure for distributed counting
- Periodically, synchronizes with other peers
- Authored using the Lasp programming model (PPDP '15)

Specialize the overlay network at runtime

- Evaluate which overlay can support the most clients
- Two evaluated: client-server vs. peer-to-peer
- Not evaluated: full-mesh (unrealistic for mobile application)



SCALING LASP, P2P KVS: TRADEOFFS

SUMMARY: IMPROVING SCALABILITY

Enables the use of actor systems for larger-scale applications

- Different overlays enable larger number of clients
- Overlays allow more traditional communication patterns for mobile applications
- May be suitable for “Internet of Things” applications

Performance optimizations

- Supported by all topologies

Prototype

- Peer-to-peer topology adopted by community members
- Used on hardware devices in LightKone EU-H2020 project on edge computing

IMPROVING LATENCY

Techniques for latency reduction by enabling parallelism of the network.

IMPROVING LATENCY

Head-of-line blocking

- Background cluster messages for maintenance, failure detection, cluster membership, etc.
- Application-behavior blocked and/or delayed

Queueing delay

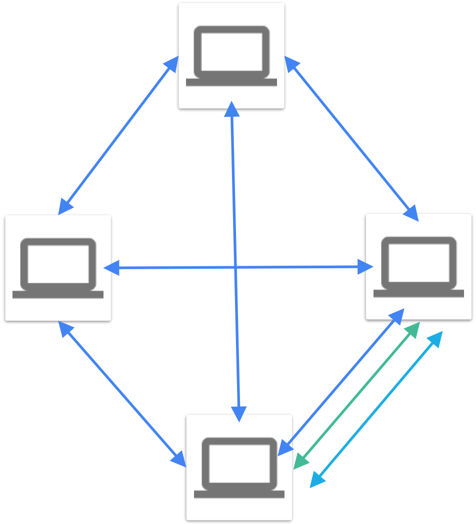
- Fast-senders vs. slow senders
- High-latency: delay in transmission, when available bandwidth for parallelism
- Large-payload: other senders are blocked during transmission and serialization/deserialization

Can we use knowledge from actors?

- Act sequentially
- Have identities and send to actors by identity

NAMED CHANNELS

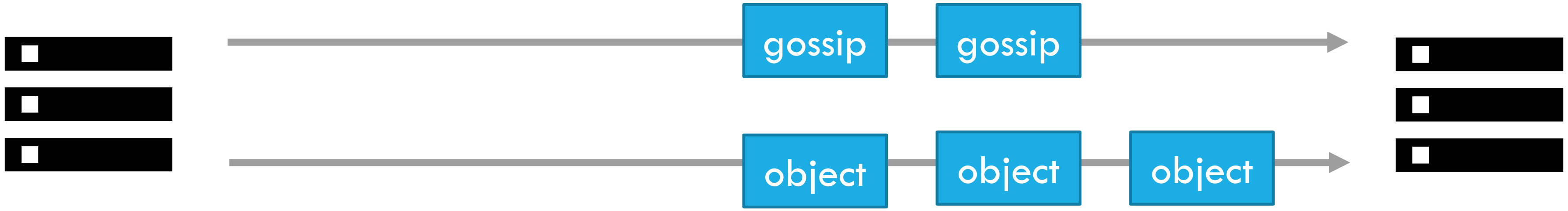
All we require is programmers to **annotate the type** of message when sending a message to another actor.



Enable multiple TCP connections between nodes for segmenting traffic.

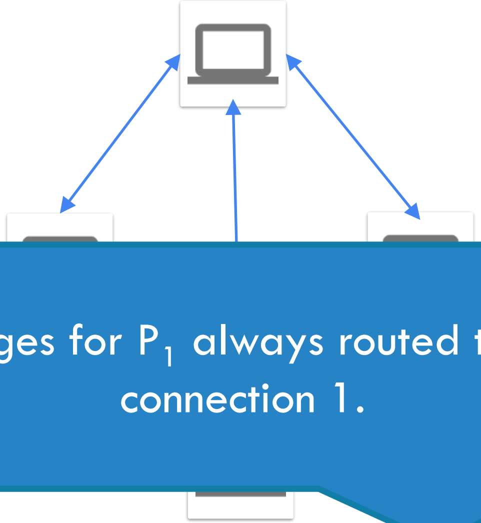
Alleviates head-of-line blocking between different types of traffic and destinations.

Beneficial for isolating background maintenance traffic from application-specific traffic.



AFFINITIZED PARALLELISM

Automatic, given process identifier **or** with an **annotation** from the programmer if using a different key.



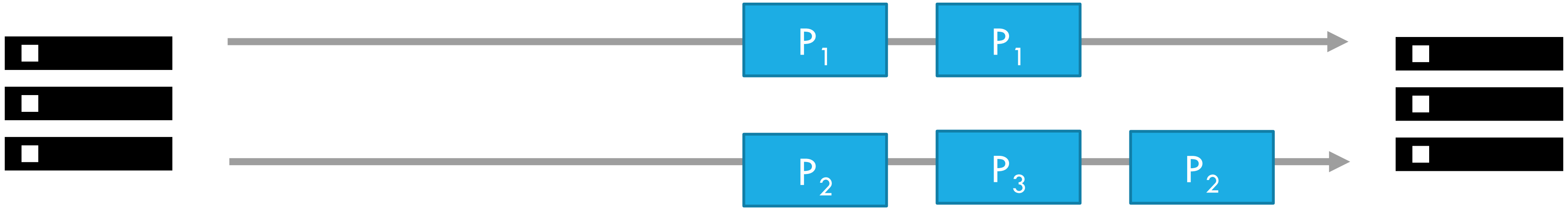
Messages for P_1 always routed through connection 1.

Enable multiple TCP connections between nodes for increased parallelism.

Partition traffic using a partition key.

- Automatic placement (using process identifier)
- Manual partitioning (using user-specified partition key)

Beneficial for separating slow-senders from fast-senders



PROGRAMMER ANNOTATIONS

Channels

- Specify channel name

Affinitized scheduling

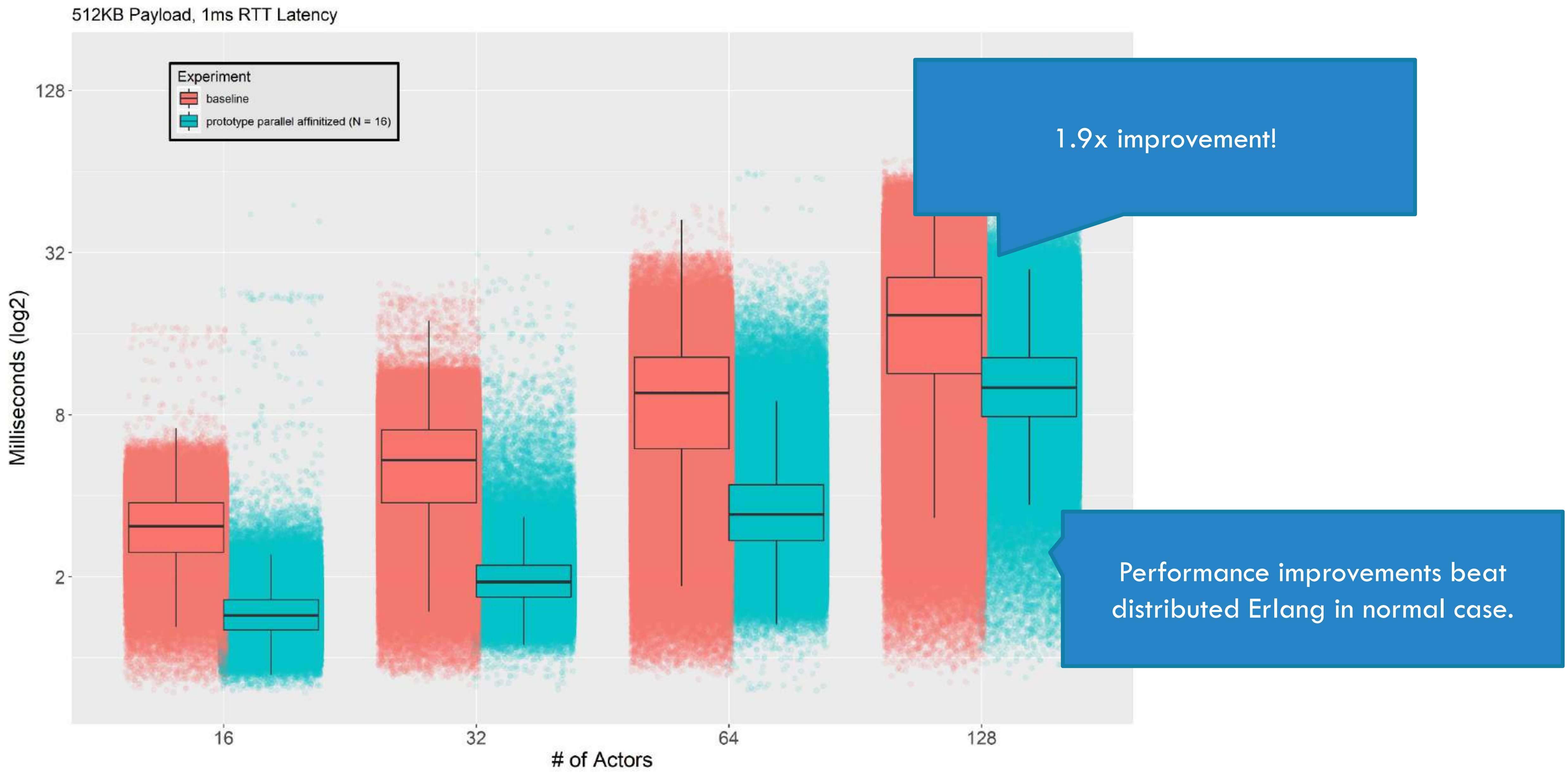
- Specify partition key

```
-import(partisan_peer_service_manager, [forward/3]).

%% Specify channel.
forward(Dst, Msg, [ {channel, Channel} ]).

%% Override key for affinity.
forward(Dst, Msg, [ {partition_key, Key} ]).
```

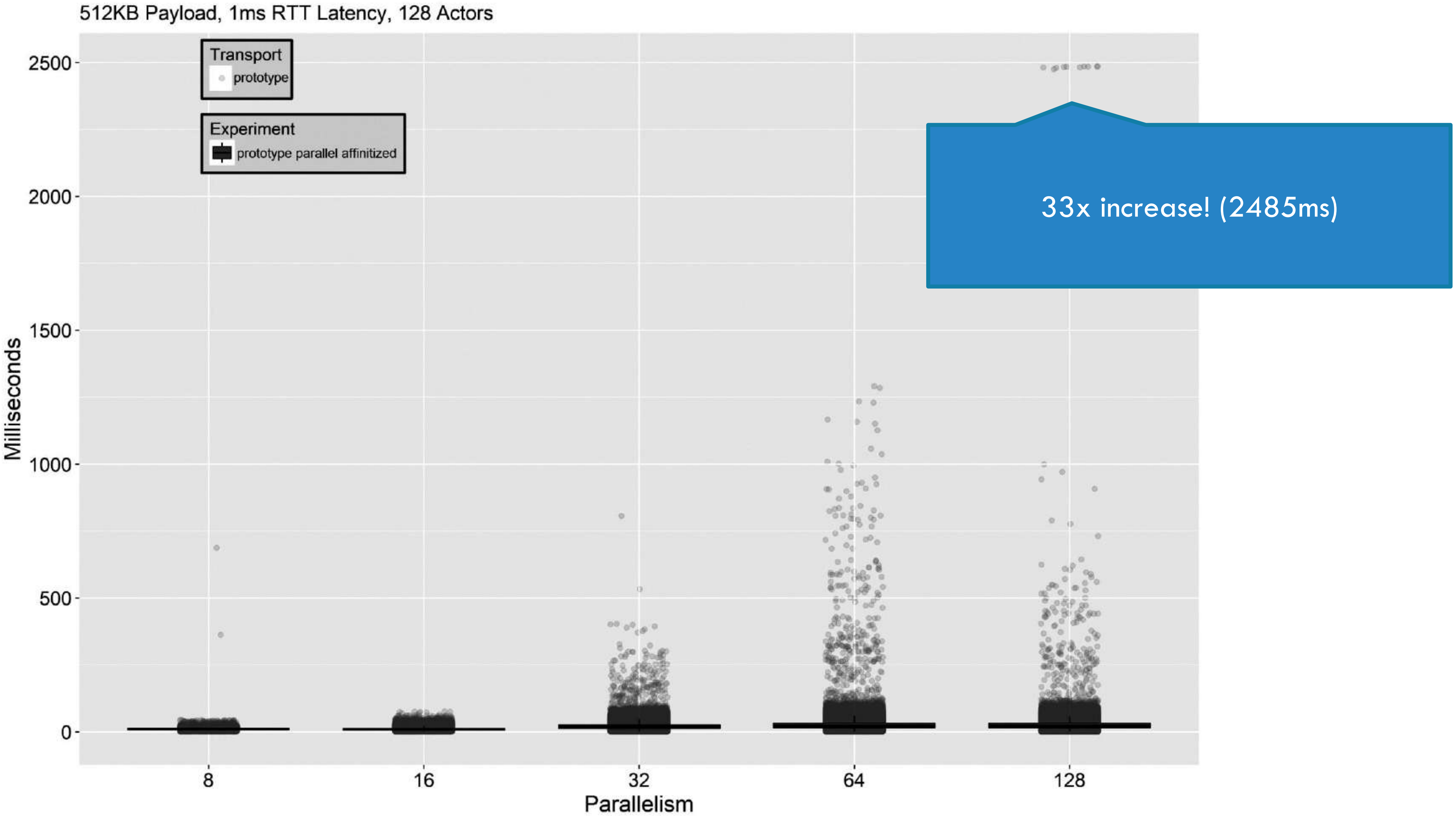
Override parameters, if necessary.

EVALUATING LATENCY

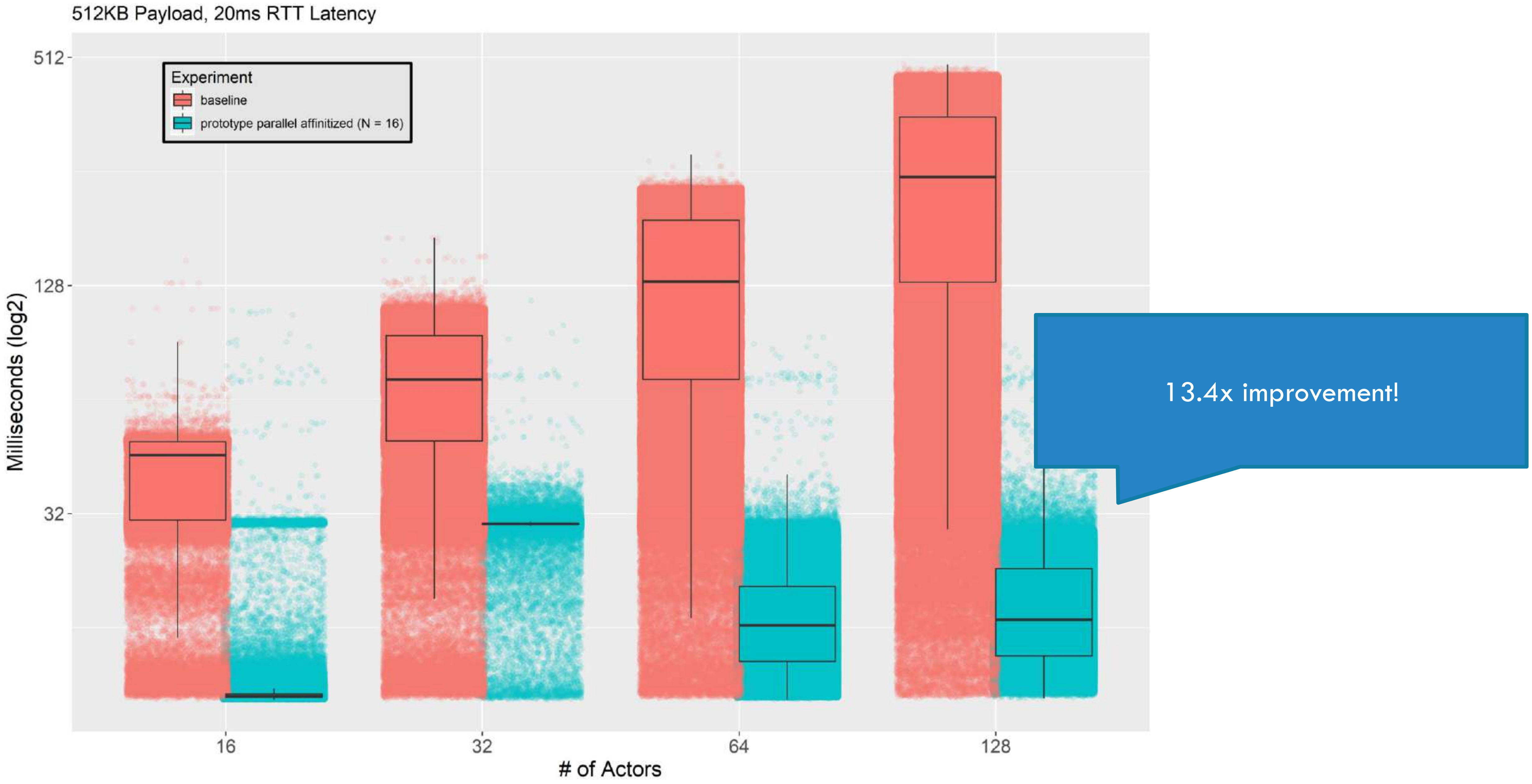
Techniques for latency reduction by enabling parallelism on the network.



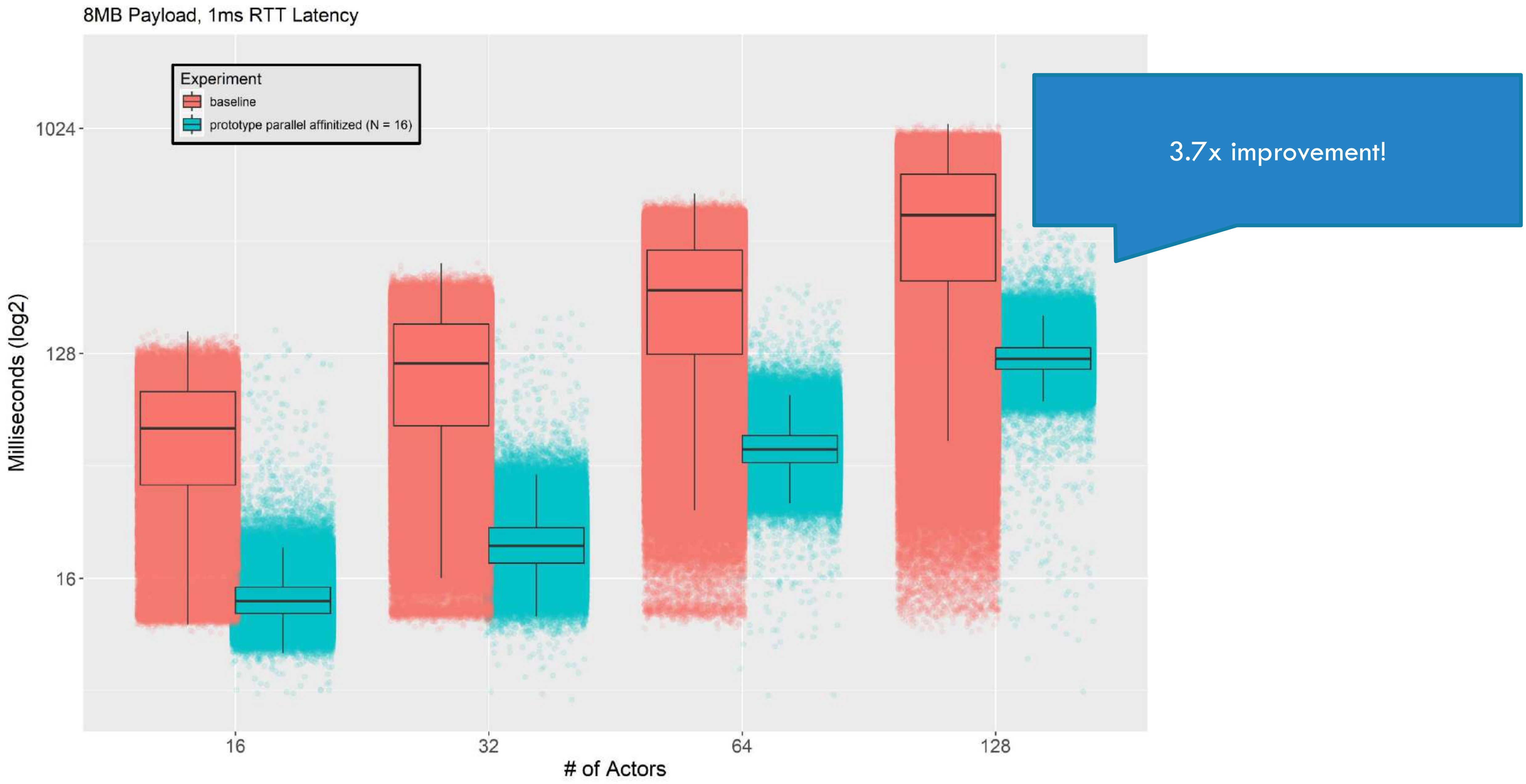
BASELINE VS. OPTIMAL PERFORMANCE: 1MS



QUEUE MAINTENANCE OVERHEAD



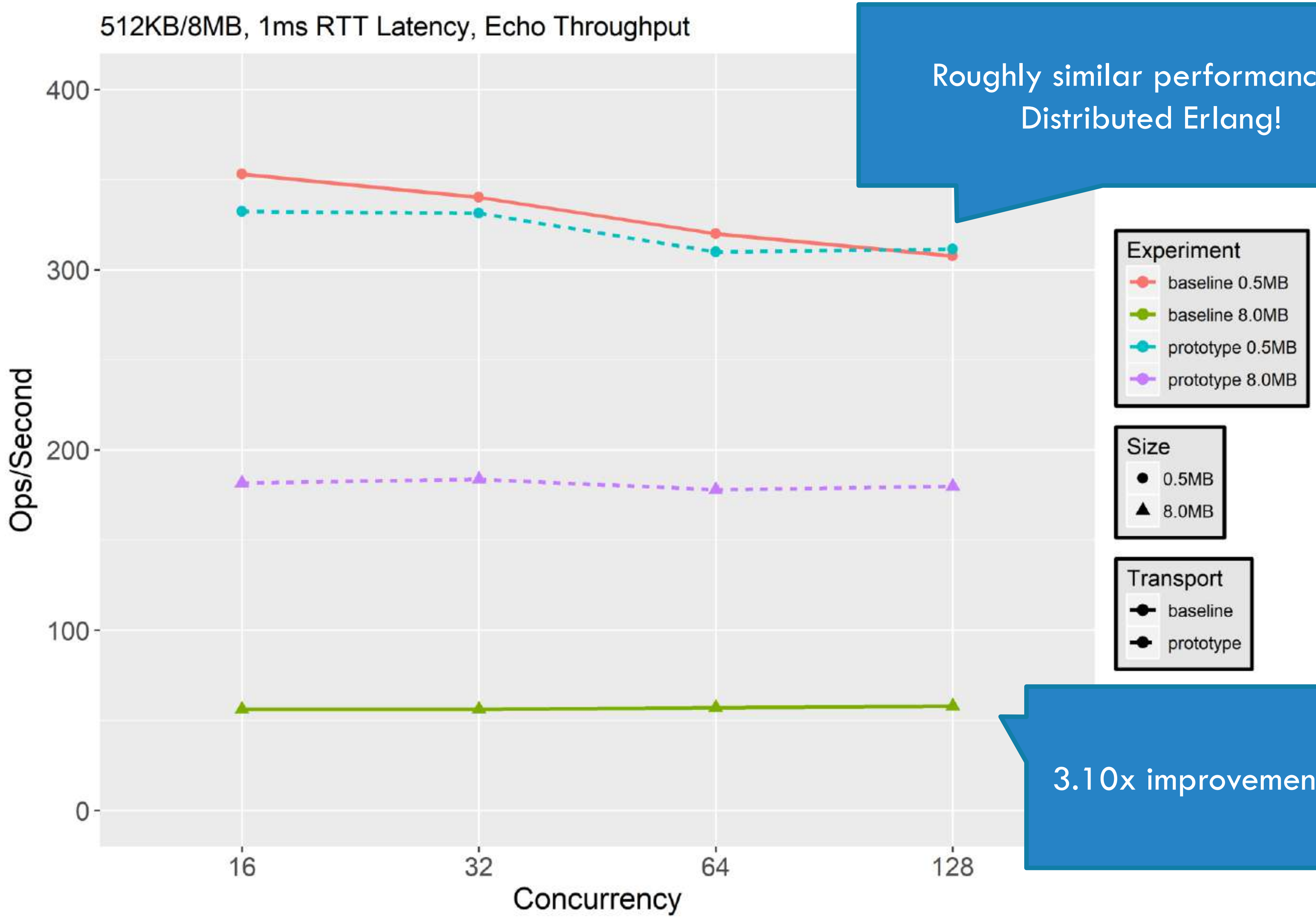
INCREASED LATENCY MESSAGING: 20MS



INCREASED PAYLOAD MESSAGING: 8MB

EVALUATING LATENCY: RIAK CORE

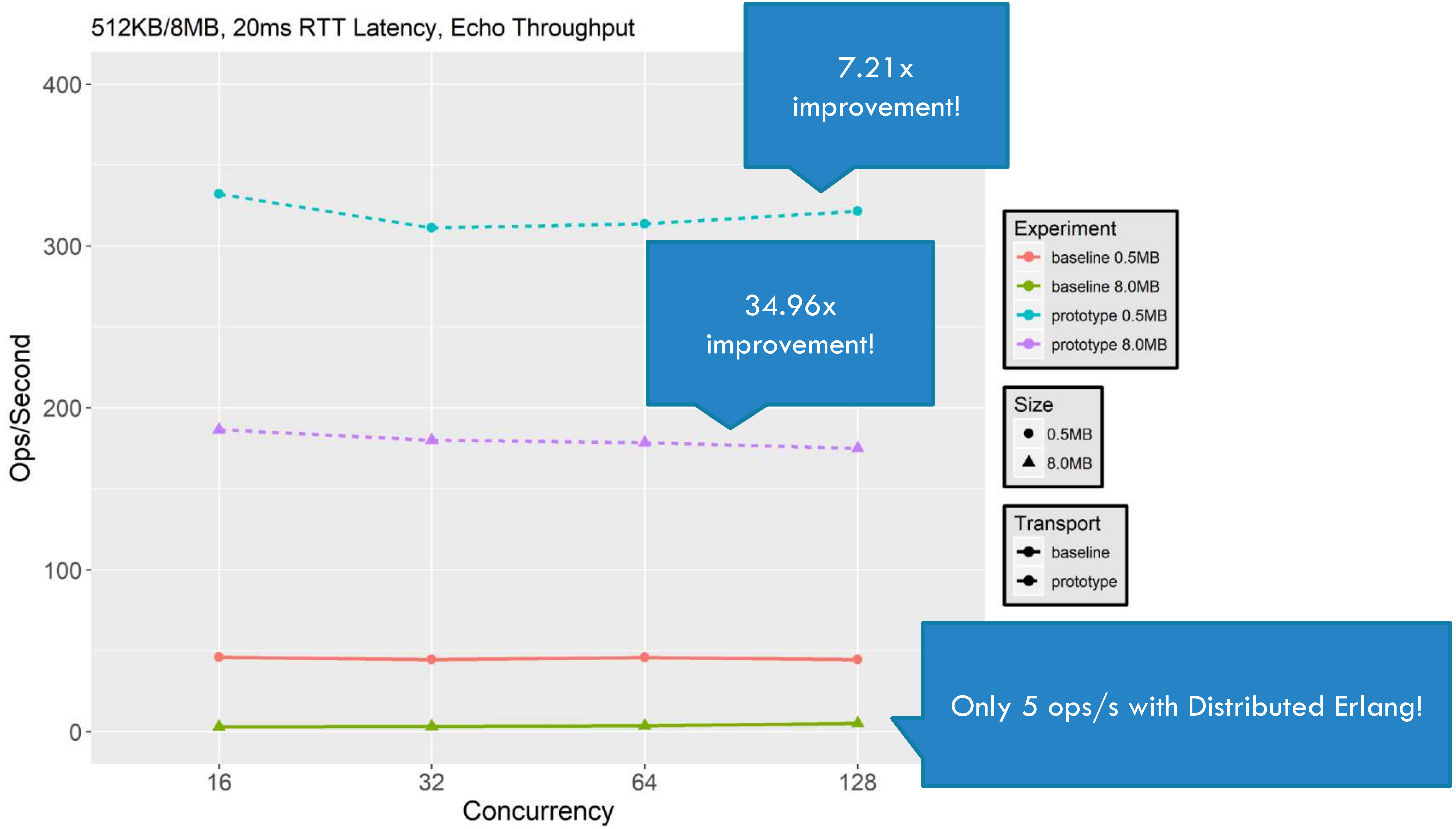
Techniques for latency reduction by enabling parallelism on the network.



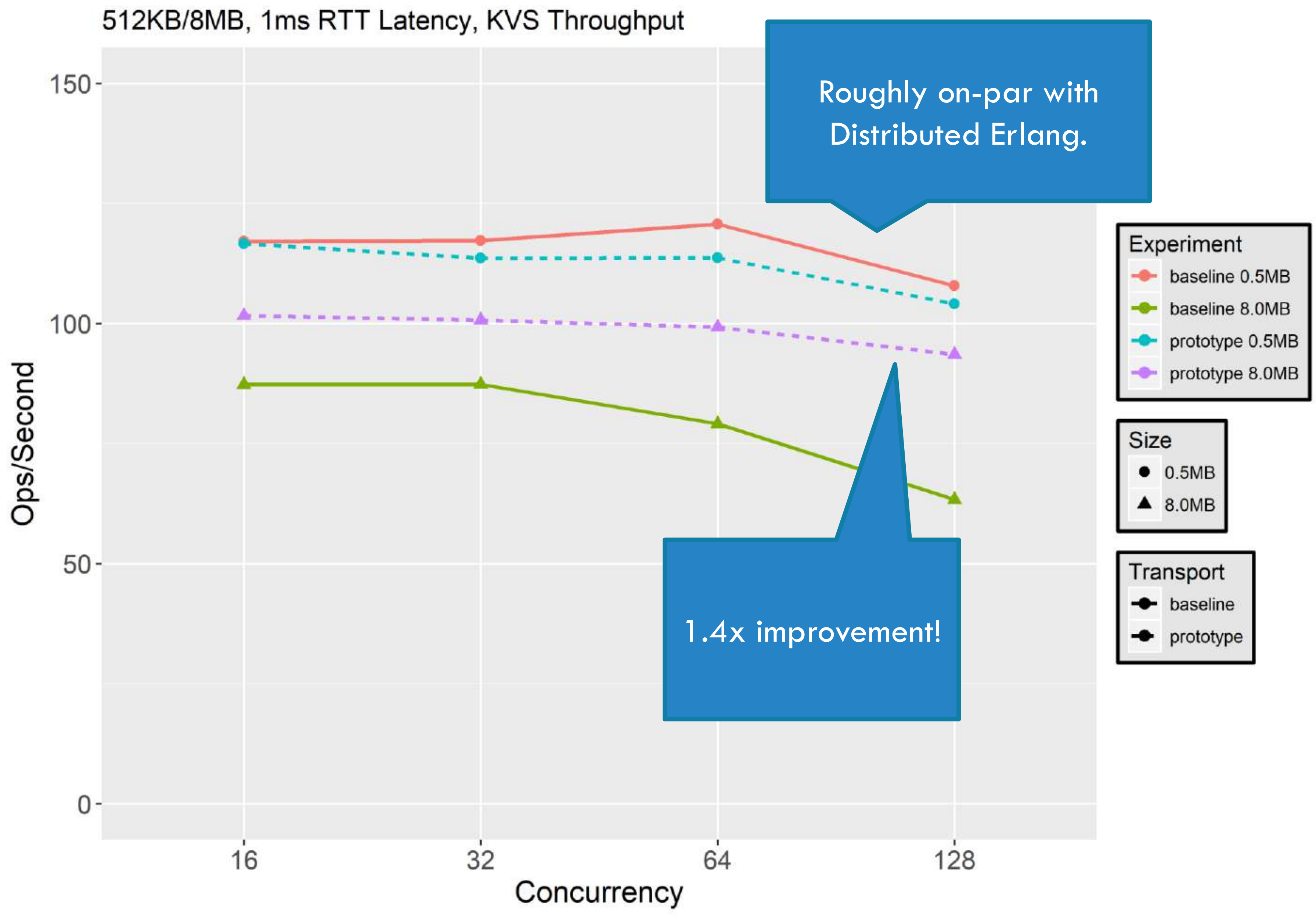
Roughly similar performance to Distributed Erlang!

3.10x improvement!

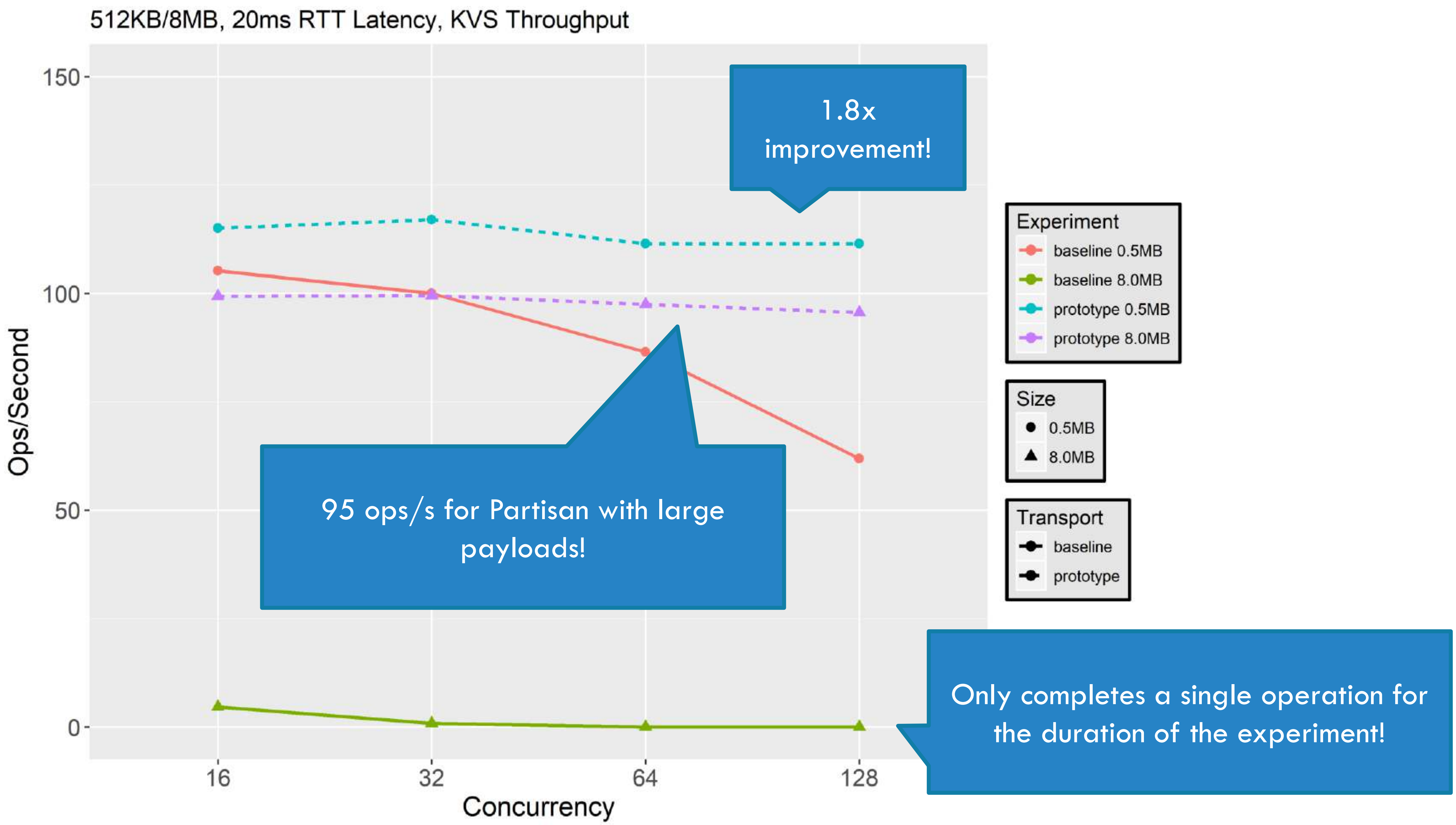
ECHO SERVICE: 1MS



ECHO SERVICE: 20MS



KVS SERVICE: 1MS



KVS SERVICE: 20MS

SUMMARY: IMPROVING LATENCY

Performance on-par with Distributed Erlang

- Can achieve similar, if not better, performance in designed case
- Distributed Erlang is designed for single AZ/region

Enable new types of applications

- Large data-centric workloads
- Geo-distributed applications (multi-AZ, possibly multi-region)
- Combination of both

Prototype

- Validated on real-world programming framework
- Some adoption of our library

PARTISAN V3

What's coming in the next version of Partisan?

PARTISAN V3 IMPROVEMENTS

Membership Strategies

- DSL for implementing membership protocols
- 3 implementations: Scamp, HiScamp (in progress), Cyclon
- Connection maintenance is automatic, user only has to handle membership events

Orchestration

- Auto-clustering using Mesos, Docker Compose or Kubernetes
- Partisan will automatically discover peers and cluster them

Example Applications

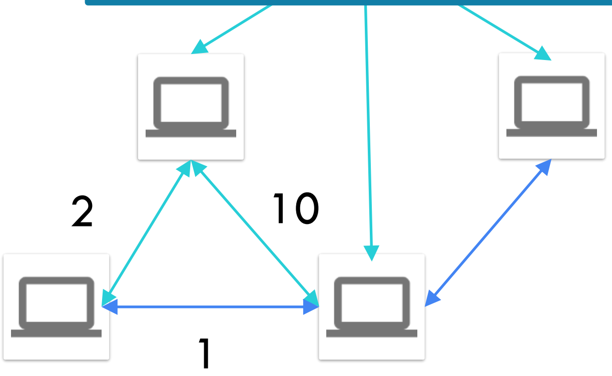
- 2PC, 2PC+CTP, 3PC, Gossip (3 variants)

Performance and bugs fixes.

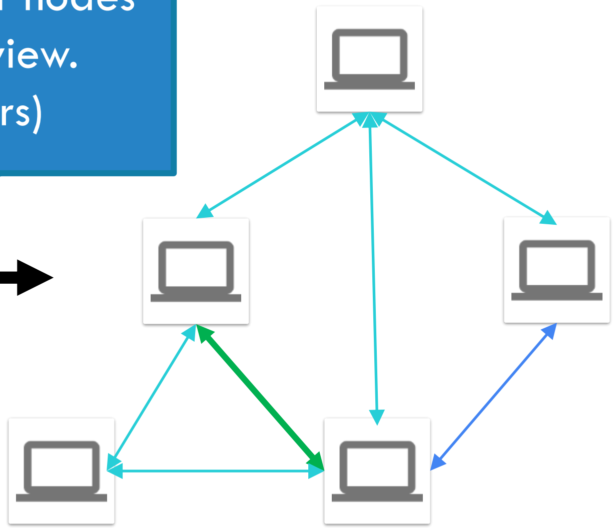
- Many performance improvements and bug fixes

X-BOT: ORACLE OPTIMIZED OVERLAYS

4-step optimization pass for replacement of nodes in the active view with nodes in passive view. (for random selection of active members)



Not all links have equal cost – with cost determined by outside "oracle."



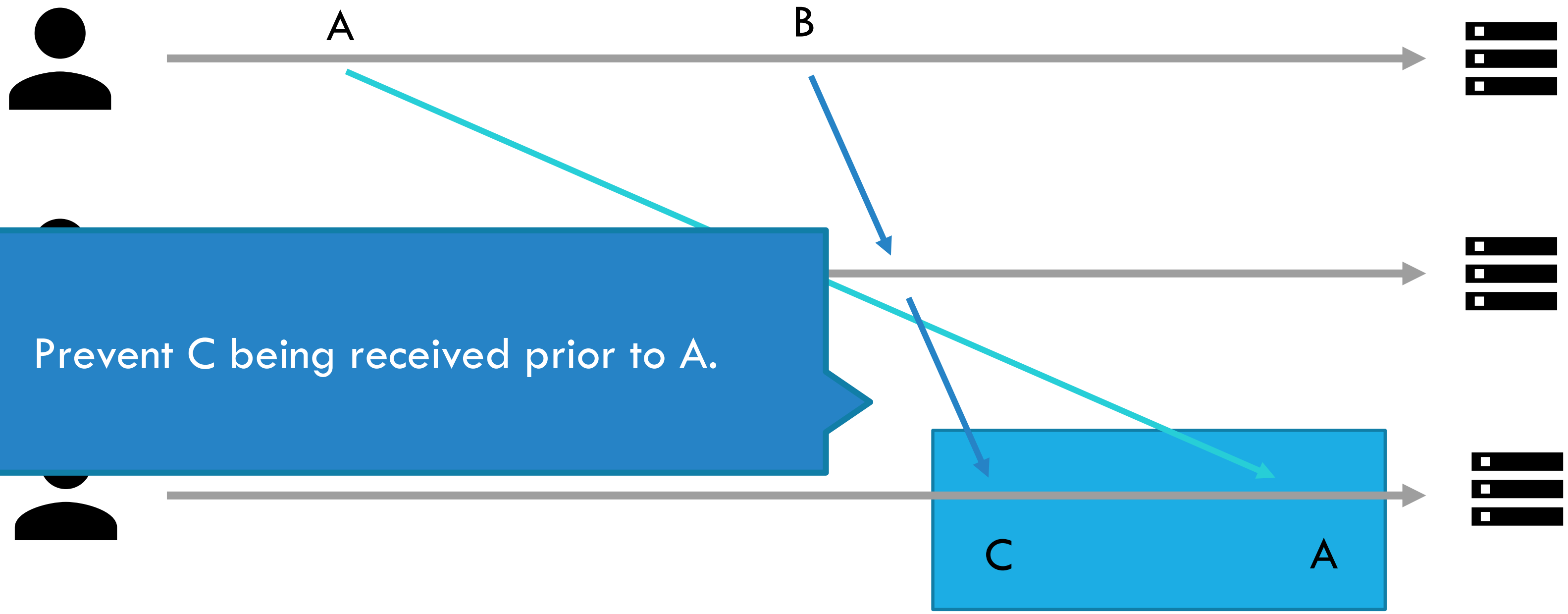
Reduce dissemination latency by optimizing overlay accordingly – swap passive and active members.

CAUSAL ORDERING

Important for overlays where message might not always take the same path!
(ie. HyParView, etc.)

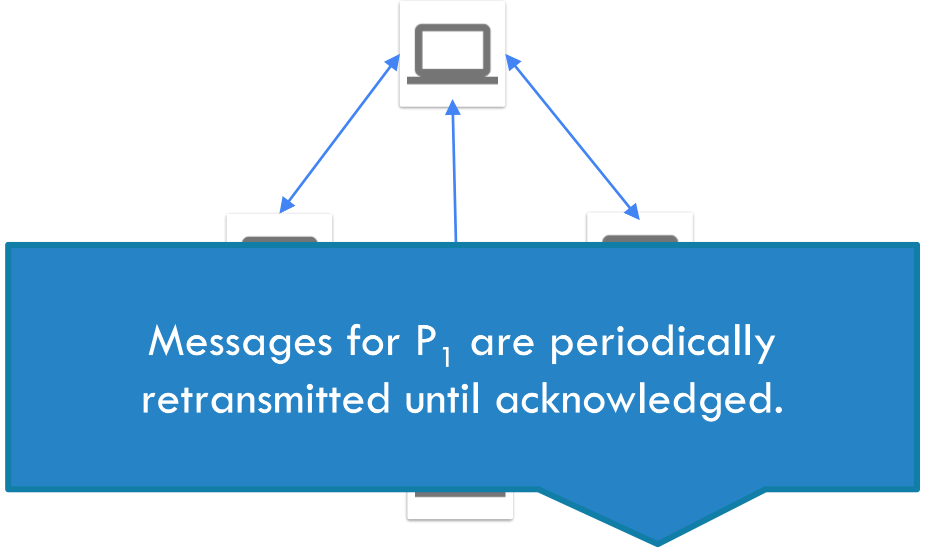
Ensure messages are delivered in causal orde

- FIFO between process pairs of sender/receiver
- Holds transitively for sending and receiving messages



Prevent C being received prior to A.

RELIABLE DELIVERY

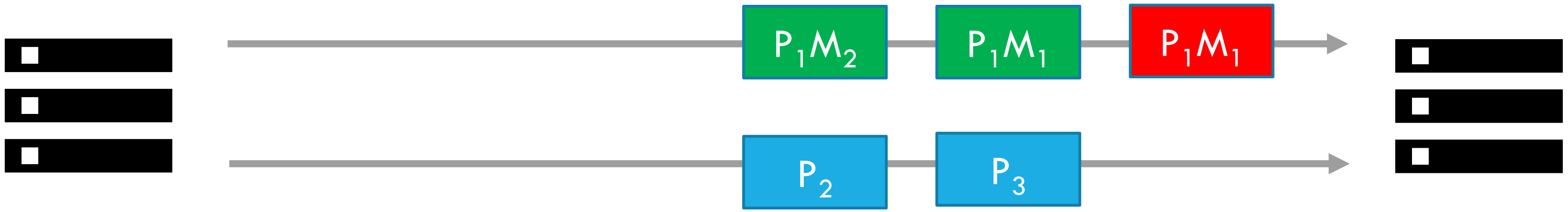


Buffer and retransmit messages using acknowledgements from destination

Per-message or per-channel

At-least-once delivery (to the application)

Needed for causal delivery where a dropped message might prohibit progress



MESSAGE INTERPOSITION

Incoming Message

Pre-Interposition

Interposition

Post-Interposition

Actor

Message Processing Pipeline

Reorder messages.

Rewrite message content or drop message.

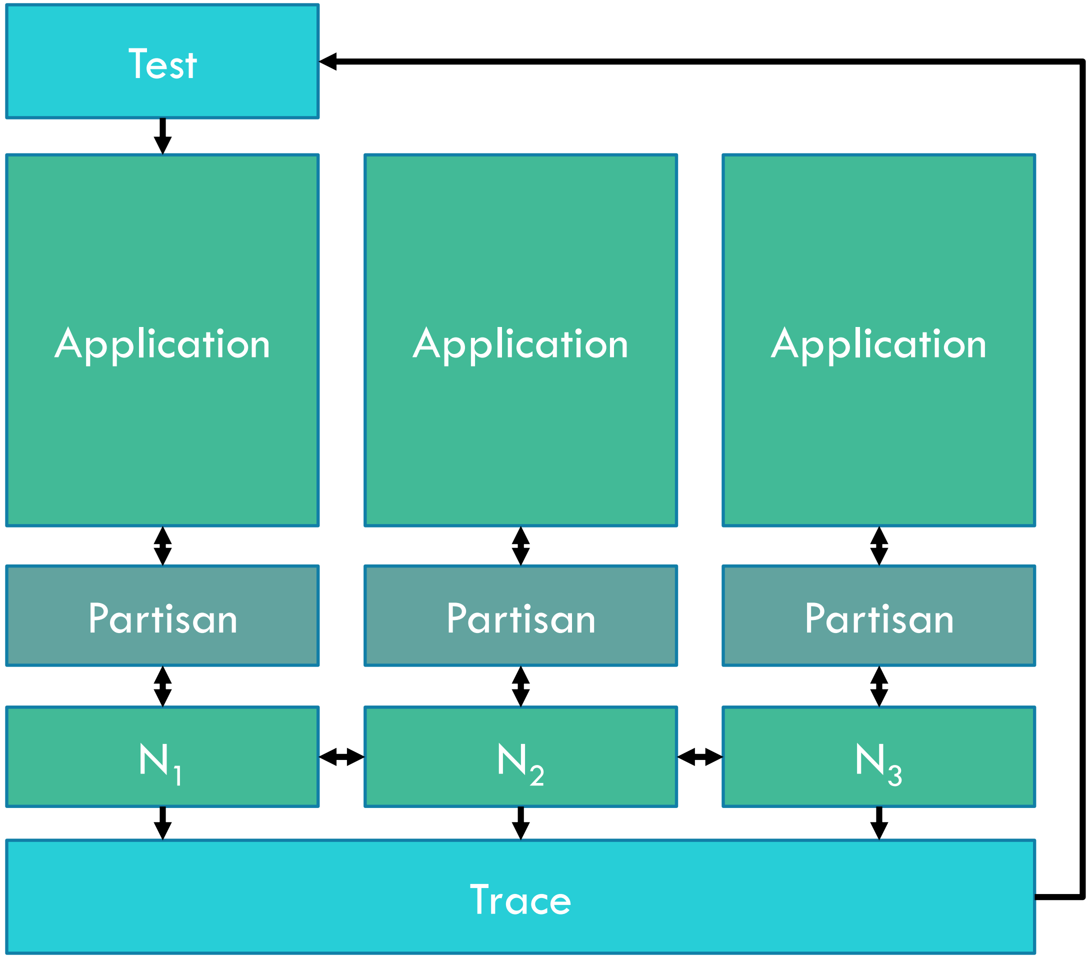
Record result of message processing.

Dynamic instrumentation, enabled at runtime.

TRACING, DEBUGGING AND REPLAY

Start with unit test.

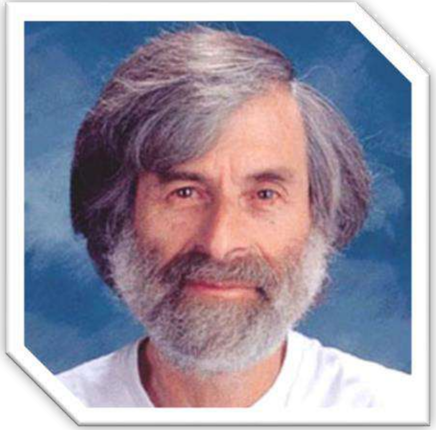
Deterministic replay of execution.



Generate trace of execution.

Distributed applications are ubiquitous, everyone’s writing them!

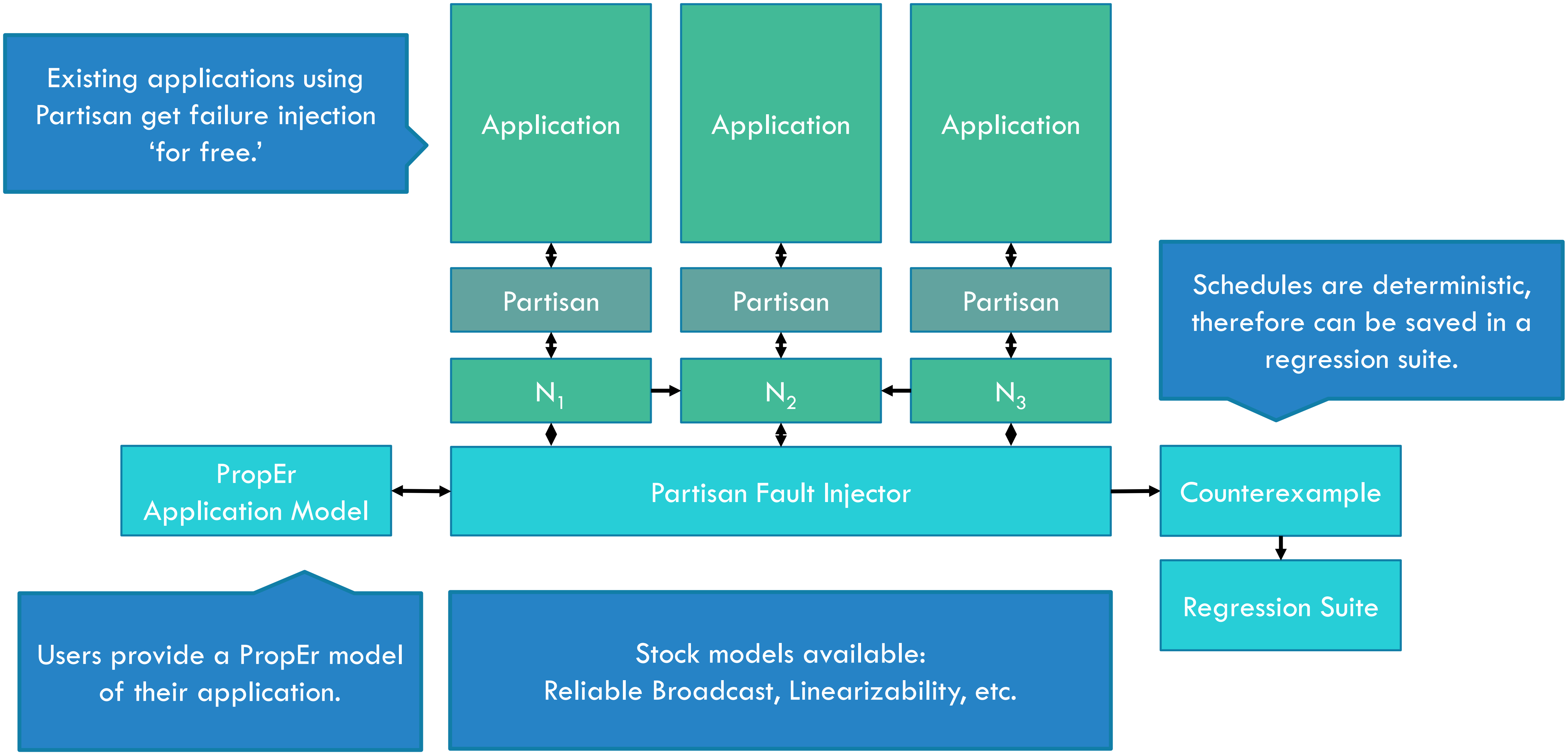
However, distributed applications are still very difficult to write because servers can crash and messages can be lost!

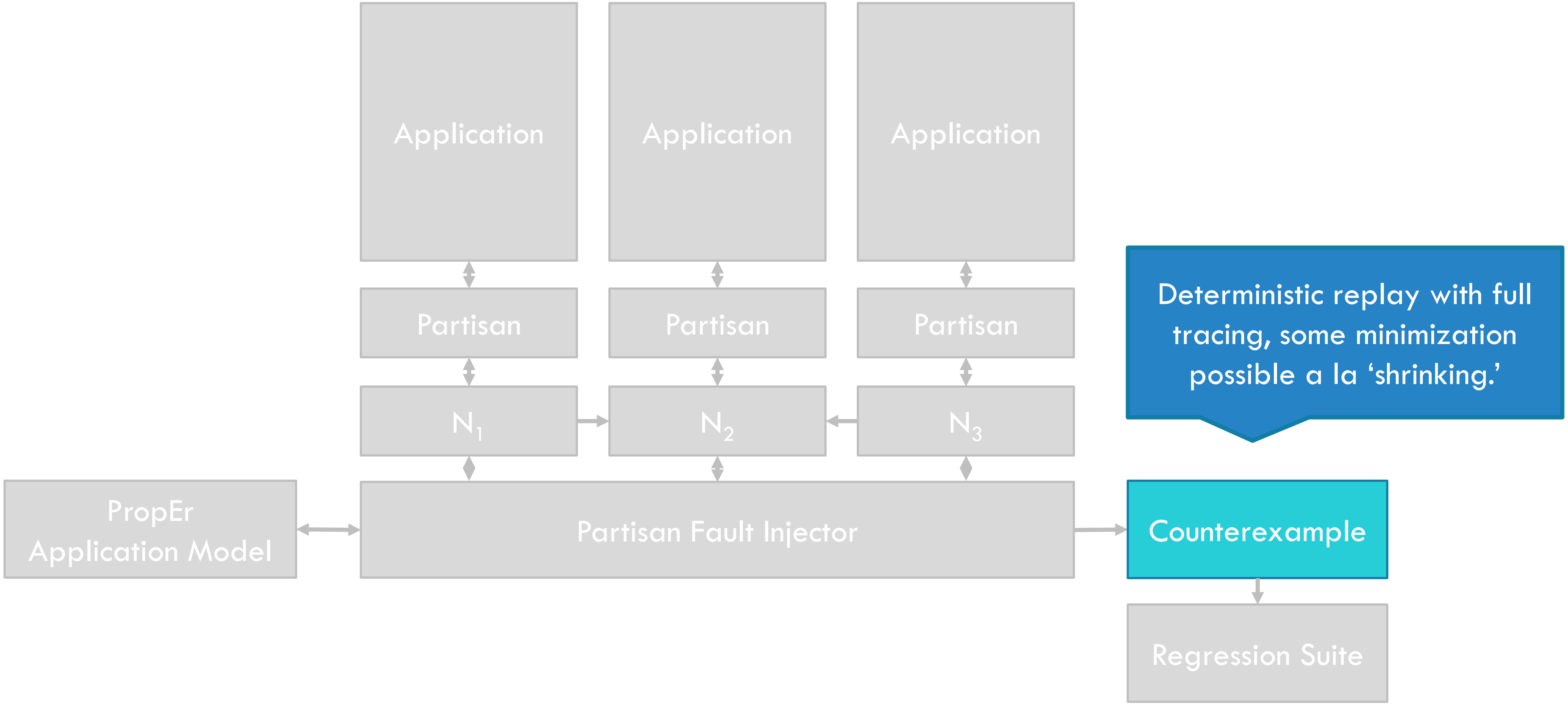


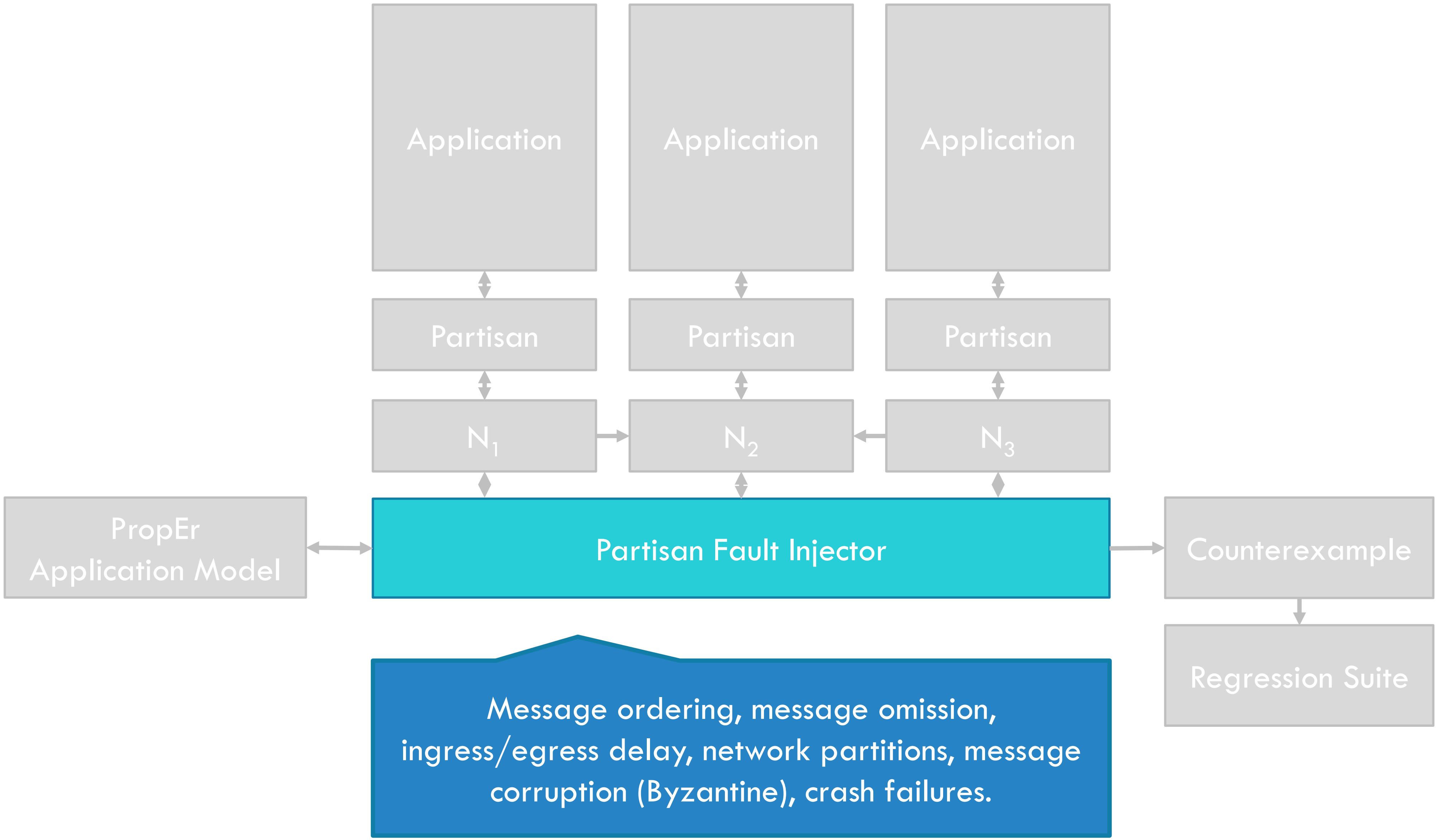
Do you know what your application will do if a **message is lost**? What if the application server **crashes** in the middle?

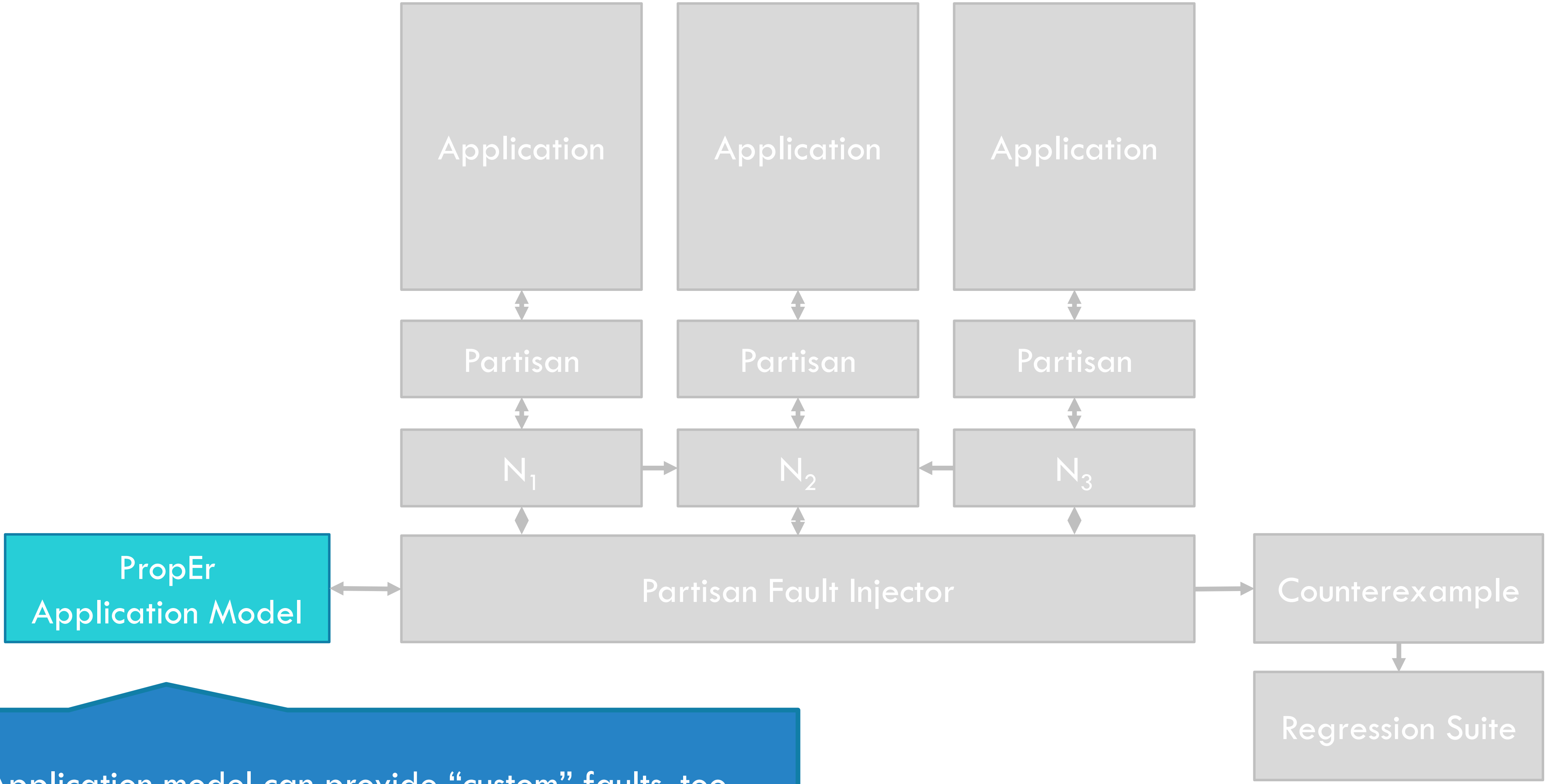
VERIFYING RESILIENCE

Verifying your application runs correctly under failure.

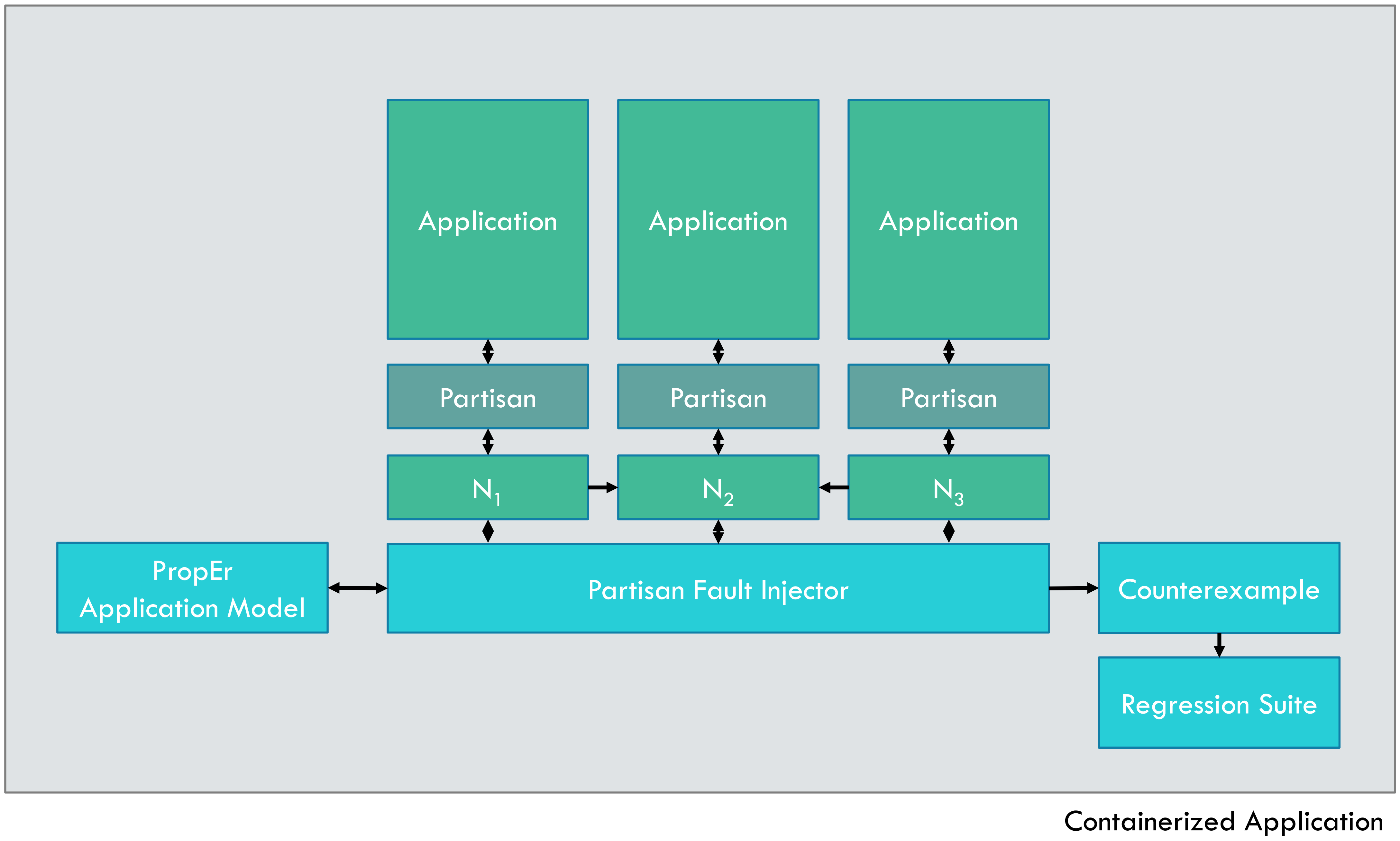

Existing applications using Partisan get failure injection 'for free.'
Application
Application
Application
Partisan
Partisan
Partisan
N1
N2
N3
Schedules are deterministic, therefore can be saved in a regression suite.
PropEr Application Model
Partisan Fault Injector
Counterexample
Regression Suite
Users provide a PropEr model of their application.
Stock models available:
Reliable Broadcast, Linearizability, etc.

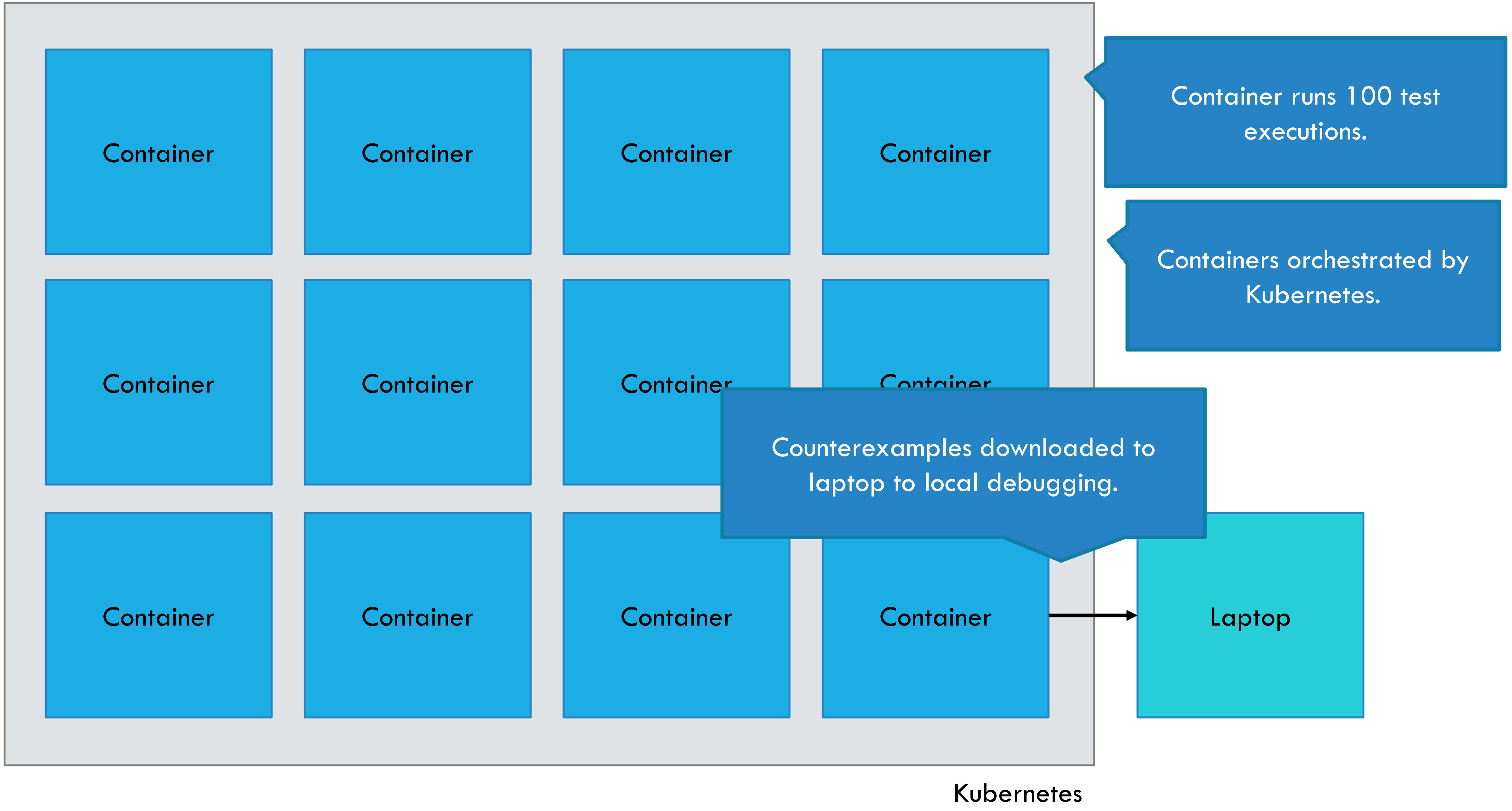

Application
Application
Application
Partisan
Partisan
Partisan
N_1
N_2
N_3
PropEr
Application Model
Partisan Fault Injector
Deterministic replay with full tracing, some minimization possible a la 'shrinking.'
Counterexample
Regression Suite

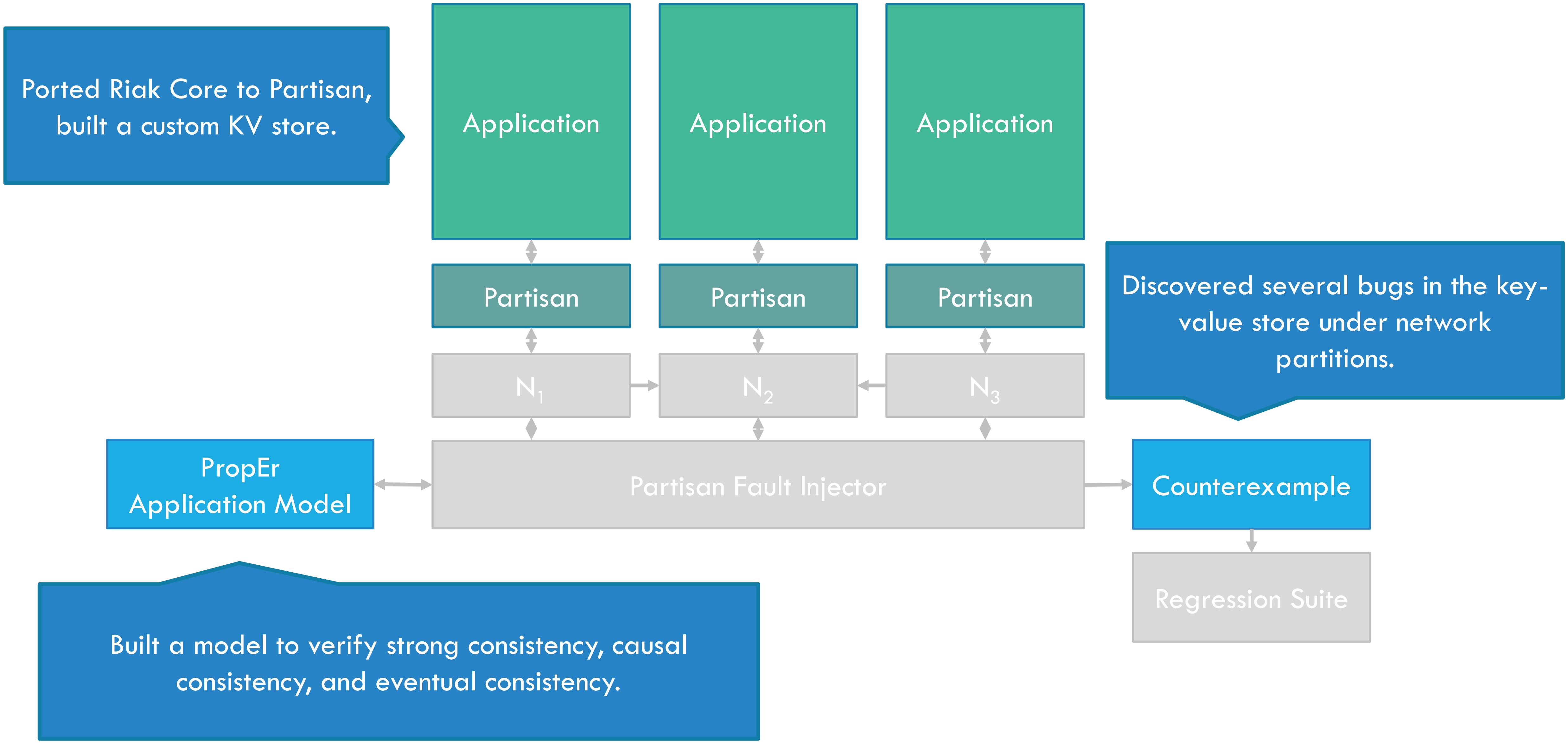

Application
Application
Application
Partisan
Partisan
Partisan
N_1
N_2
N_3
PropEr Application Model
Partisan Fault Injector
Counterexample
Regression Suite
Message ordering, message omission, ingress/egress delay, network partitions, message corruption (Byzantine), crash failures.



Application model can provide “custom” faults, too.
e.g. Riak: disk loss, bit flips, etc.


Application
Application
Application
Partisan
Partisan
Partisan
N1
N2
N3
PropEr
Application Model
Partisan Fault Injector
Counterexample
Regression Suite
Containerized Application


Container
Container
Container
Container
Container runs 100 test executions.
Containers orchestrated by Kubernetes.
Container
Container
Container
Container
Counterexamples downloaded to laptop to local debugging.
Container
Container
Container
Container
Laptop
Kubernetes


Ported Riak Core to Partisan, built a custom KV store.
Application
Application
Application
Partisan
Partisan
Partisan
Discovered several bugs in the key-value store under network partitions.
N_1
N_2
N_3
PropEr Application Model
Partisan Fault Injector
Counterexample
Regression Suite
Built a model to verify strong consistency, causal consistency, and eventual consistency.

PropEr Application Model

Model contains incorrect assumption that all reads should return the value of the most recent successful write.

```
join n1 n2 : OK
join n1 n3 : OK
n1 write x 1 : OK
n1 read x 1 : OK
n1 read x 1 : OK
fault partition { n1 } { n2 n3 }
n1 read x 1 : TIMEOUT
```

Counterexample 1:
Partition causes quorum unavailability.

```
join n1 n2 : OK
join n1 n3 : OK
fault egress_delay n2 n1
fault egress_delay n3 n1
n1 write x 1 : TIMEOUT
fault resolve_all
n1 read x 1 : 1
```

Counterexample 2:
Unacknowledged write visible because of timeout.

```
join n1 n2 : OK
join n1 n3 : OK
n1 write x 1 : OK
fault byzantine n2 bitflip
n1 read x 1 : FAIL, {1, -1}
```

Counterexample 3:
Parity bit flip error at node 2 returns disagreeing value.

Will random execution find all of the failures in my application?

2PC has only **one failure case,**
manifesting itself in 3 schedules out of 4,096 possible schedules.
(This considers possible omissions, not reorderings!)

3PC has one failure case that appears in a few schedules.
(From a total of 216,522 schedules, not considering reorderings.)

In short, no. To do that, we would need to search the entire execution space systematically.

But, we're working on this too!
So, stay tuned!

CONCLUSION

Bringing it all back home.

CONCLUSION

Runtime system for improved scalability and reduced latency for distributed actors

- Prototype implementation with adoption in Erlang
- Uses techniques of parallelism, affinitized scheduling, and named channels
- Specialization of overlay network at runtime without change to semantics

Performance and Scalability

- Up to 34.9x improvement in throughput
- Up to 13.4x reduction in latency
- Order of magnitude in cluster size

Partisan v3

- Coming soon, new overlays, fault-injection, bugs fixes, and more!

Thanks for coming!

You can find me on twitter at **@cmeik!**

Have any questions?

Come and talk to me about
how I can help your company
with high performance distributed
Erlang and fault-injection!

🎉👏👍😎✌

Carnegie Mellon University

Baskin Engineering UC SANTA CRUZ

isr institute for SOFTWARE RESEARCH