ACTORS FOR THE INTERNET OF THINGS

by

Arjun Shukla

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2021

BOISE STATE UNIVERSITY GRADUATE COLLEGE

PROPOSAL ACCEPTANCE

of the thesis submitted by

Arjun Shukla

Thesis Title: Actors for the Internet of Things

Date of Final Oral Examination: 25th June, 2021

The following individuals read and discussed the thesis submitted by student Arjun Shukla, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Amit Jain, Ph.D.	Chair, Supervisory Committee
Catherine Olschanowsky, Ph.D.	Member, Supervisory Committee
Casey Kennington, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Amit Jain, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGMENTS

The author wishes to express gratitude to his parents Laura and Ashish, and to his aunt Arti, for their love and support. He is thankful to Dr. Amit Jain for his guidance and to Boise State's Department of Computer Science for funding his research.

ABSTRACT

The actor model is a model for concurrent computation, centered around message passing between entities in a system. It is well suited for distributed programming, due to its semantics including very little guarantees or assumptions of reliability. Actor model implementations have grown more widespread in many languages.

The library Akka (written in Scala) is one of the most popular actor libraries. However, Akka is missing some key features. Our goal is to create our own actor library called Aurum, which not only has these features but exhibits higher performance. The new features include easy ways to forge references, configure and launch clusters, message type translations, and the ability to inject message drops and delays into every part the application. Aurum will be implemented in Rust, a programming language designed for high performance, asynchrony and high levels of abstraction that is well suited for IoT devices.

Our results show that Aurum outperforms Akka. In our benchmarks, a single server running Aurum gives three times the throughput as an equivalent Akka server, while maintaining good programmability and having features useful for IoT.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The actor model of concurrent computation [1] has lately received greater attention for its potential to take advantage of multiple processors on a single machine, and the simplicity of translating algorithms to a distributed environment. In this model, actors are threads that have sole access to their memory space. They may modify a private state, but can only interact with other actors through asynchronous passing of immutable data. Actors process messages serially, which are stored in a queue (or "mailbox") until then. Similar to network communication semantics, actor messages operate with at-most-once delivery. There is no guarantee messages are received in the same order they are sent, even from the same source actor. Fault tolerance comes naturally to actors, who may start, terminate, and monitor other actors. Actors send messages via actor references, learned about from other messages or spawning that actor. References to actors are location transparent: message delivery semantics do not change with locations of target actors. Scalability is now very simple, as deployment to multiple nodes does not necessitate any changes to existing code.

The actor model may be ideal for implementing applications within the Internet of Things [2]. To this end, and for ease of development, a systems-level programming language implementation of the actor model is needed, so actors may conveniently interact with physical devices connected to the machine they are housed on. IoT devices may be sending huge amounts of raw data to servers in charge of them. To cope with high load, our implementation must be performant, with minimal runtime overhead such as garbage collection, just-in-time compilation, exception handling, or marshalling between memory spaces of local actors. IoT environments require quick start-up time, since crashes may happen often.

Ideally, an actor model would be type-safe. This means whether an actor may receive a certain type of message or not is decided at compile time, leading to safer programs. References to actors needed for sending messages should be forgeable (i.e. they can be spontaneously created). Type safety adds new complexities to an actor model. A good implementation will make these inconsequential or at least easy for the programmer to deal with.

1.1 Existing Actor Model Implementations

Some popular actor libraries already exist, but none fill the niche we are looking for. In Erlang [3], the language and BEAM VM itself are built around actors, but may suffer from poor performance due to its dynamic typing, interpretation and its isolation of memory spaces for local actors. Erlang is a dynamically typed language, and does not provide the type safety we are looking for. Akka is an actor model written for the Java Virtual Machine (JVM). It's performance suffers from the JVM's just-in-time compilation and garbage collection. Akka does not come with the same memory safety guarantees as Erlang, as messages to local actors are not copied, but only their references. In cases where mutable objects are being transferred, location transparency breaks down. In this situation, there is now a mutable, shared state among local actors. Only serialization and deserialization provide a guaranteed deep copy in Java, the clone() method in Java is shallow by default. Serialization in local messaging can be enabled in Akka [9], but the performance benefit from sharing memory is lost. In Java, serialization is a run-time check, which decreases our type safety.

1.2 Thesis Statement

Can we develop a type-safe implementation of the actor model that is safer to use in developing distributed solutions and is more performant than existing actor model implementations? The use case of particular interest is IoT devices. We will compare our performance to the Akka actor model.

1.3 The Rust Programming Language

Rust [10] is a systems language, sufficiently low-level to readily interact with local devices. It incorporates extensive compile-time checks to ensure memory safety. The compiler prevents dangling pointers, most memory leaks and data races through keeping track of which thread in what scope owns a piece of memory, for both the stack and the heap. Leveraged properly, this could function as an effective replacement for marshalling data across local actors: many implementations of the actor model create explicitly separate memory spaces, where copying is necessary. Rust's memory management is entirely handled at compile time, memory is freed when owned references to it fall out of scope. Lack of runtime garbage collection is likely to significantly improve performance. Instead of exceptions, Rust handles errors through matching patterns on return values. Determining control flow with return values is much faster than traversing through the stack. All of Rust's executable

Table 1.1: Feature comparison between Rust, Scala and Erlang

Feature	Rust	Scala	Erlang
Asynchrony	Built-in	With Akka	Built-in
Reflection	No	Yes	N/A
Safe Concurrency	Yes	No	Yes
Serialization	With Serde	Built-in	Built-in

code is generated at compilation time, minimizing program startup time, useful in situations where crashes and restarts happen often.

We will compare Rust to 2 other programming languages popular in distributed computing: Scala and Erlang. Akka is implemented in Scala and meant to be used with Scala (but also has a Java API). Erlang has actors built into the language. Table 1.1 compares language features of Rust, Scala and Erlang relevant to the actor model. The Rust compiler's treatment of Futures turns them into efficient state machines that can be managed by executors and schedulers. Rust's lack of reflection is problematic when trying to implement distributed actors, but a workaround is implemented in the library. Rust tracks mutability through reference types. It ensures safe concurrency using the Send and Sync traits. Serde is a Rust library with traits and macros to aid serialization. Rust checks trait implementations at compile time, so no runtime errors related to nonserializable types can occur.

1.4 Other Actor Systems for Rust

Multiple software projects are currently implementing the actor model in Rust. Actix [7] is the most well-developed. It is mature and fully functioning, but it only allows local actors. There exists young plugin called actix-remote, but it is underdeveloped and does not implement remote actors for actix. The bastion-rs [5] project was created with the goal of creating an Erlangesque actor interface for Rust, but it is meant for small-scale distributed systems according to their website. Bastion also uses untyped, unforgeable actor references, does not handle serialization or deserialization, does not support static dispatch and cannot send remote messages outside the cluster. The Riker [4] project was created to mimic the popular Akka library for Scala [8]. While Riker looks like it might develop into a Akka-for-Rust library eventually, many necessary features remain unimplemented such as remote actors, location transparency and clustering. Lastly, Acteur [6] is a fairly new project, and does not support remoting or clustering. None of these libraries are sufficient to use out of the box for the functionality we want.

1.5 Background

We assume some knowledge of distributed systems and concurrency, including servers, clients and threading. *Distributed Systems* by Martin van Steen and Andrew Tanenbaum [29] is a good resource for learning the principles of distributed systems. The book can be downloaded and read for free.

CHAPTER 2

LIBRARY FEATURES AND DESIGN

Our library will be called Aurum, named after a group of alien robots from the video game *Kid Icarus: Uprising*. This section will cover various features of Aurum and how they are implemented.

2.1 Local Concurrency

Logically speaking, each actor runs on its own thread, and messages are received atomically. Actors must be lightweight and have good scalability locally, meaning system threads are not a good choice for running actors. Tokio is a Rust library with a runtime environment for scheduling green threads (which Tokio calls “tasks”). It takes least one Tokio task to run an actor. Tasks are lightweight, and many actors can be run on a single system thread. Tokio integrates with the Rust compiler’s asynchronous features to schedule its tasks. Aurum operates under the same rules as Rust’s (and Tokio’s) asynchrony: tasks are IO-bound. Code running in tasks is expected to `await` often, and starvation can occur if asynchronous code performs heavy compute operations or synchronous IO. There are plans to add support for heavy compute to Aurum in the future.

2.2 Location Transparency

There are 2 types of actor references, LocalRef and ActorRef. LocalRef is not serializable and accepts messages that implement Rust's Send trait. A LocalRef directly places messages in the message queue of the recipient actor. A LocalRef type is needed because some message types may not be serializable. ActorRefs do not accept non-serializable messages, although it is possible for ActorRefs expecting non-serializable message types to exist. All ActorRefs, even ones expecting non-serializable messages, are serializable. They contain information needed to contact a remote actor: its location, port and name. ActorRefs may also contain a LocalRef if they refer to a local actor. Sending a message with an ActorRef can make use of the inner LocalRef if it exists. ActorRefs are location transparent: its choice between local and remote sends occur without the user needing to care.

2.3 Typed Actor References

Actor references are typed, meaning they can only receive messages of a specific type. Typed references add safety by turning some runtime errors into compile time errors, and eliminating the need for catch-all match branches that handle invalid messages. It also makes code easier to read, type annotations give extra hints of the actor's role in the program. Actor references are statically typed, meaning the type of messages they receive must be defined at compile time. Generics are used to communicate which the types of messages.

ActorRefs can be serialized and deserialized. When a deserialized ActorRef is loaded into the structure, we need to make sure the ActorRef actually refers to an actor with the same message type the compiler thinks it has. Type information about actor references must be serialized and sent along with the rest of the ActorRef. When the type information is deserialized, a check must be performed to make sure the actor reference is valid for the type it is being loaded into. In other words, we are asking a piece of data about its type information at runtime (known as reflection). Since Rust does not have reflection, Aurum deals with this problem by keeping a centralized list of all message types used in the system in the form of an enum. Each type is represented by a different variant within this enum (henceforth referred to as the “unified type”).

Actor references must be deserialized using the right generic type, so names of actors are paired with an instance of the unified type conveying what message type the actor with that name receives. The actor reference’s message type needs to be a member of this unified type, and a single logical system must operate with exactly unified type. The unified type implements a case trait for every message type in its variants. The case trait contains associated constants, allowing us to retrieve variants of the unified type with Rust’s type information for runtime comparison.

A programmer using Aurum must define the unified type themselves. Since 2 different unified types cannot exist for the same application, libraries built on top of Aurum’s core functionality (including our clustering) cannot define a unified type. Instead they must define their logic in terms of generic types that implement case traits for their message types. The user of their library will define the enum to include all the message types for all the libraries being used.

2.4 Actor Interfaces

It is not unusual for someone to want to store actor references in a data structure (for instance, to keep track of subscribers to events). Collections in Rust must be homogeneous, but there may be many different kinds of actors who want to subscribe to the same events. You cannot keep actor references receiving different types in the same data structure, nor is creating a different structure for every possible type desirable, nor do we want to create separate actors that relay event messages to the real subscriber under its preferred type.

Actor interfaces allow the programmer to define a subset of message types receivable by an actor, and create actor references which receive only that subset. For LocalRefs receiving a message of the subtype, the reference will internally convert it to the actual type the recipient actor receives and send it off.

Remote interfaces are more complicated. Actor references have the interface as their outward-facing generic type, not the underlying message type of the recipient actor. This information is still present in the actor's name, which is important for forged references. When a message sent by an ActorRef is serialized, the interface variant is sent along with it to tell the recipient deserializer how to interpret the bytes. Based on the variant, the deserialization is dynamically dispatched to the correct interpretation.

2.5 Forgeable References

Traditionally, actor references can only be obtained 2 ways: receiving it in a message or being in charge of spawning the actor. However, it is useful to be able to create actor references from scratch by providing all the information needed ourselves. Forging references removes the need for some complex discovery protocols in distributed algorithms. This cannot be done with typed actor references in Akka, because Akka actor names do not contain any type information. Since Aurum includes type information in its actor names, it is capable of creating actor references with certain, compile-time knowledge of which messages the actor it refers to accepts. The semantic validity of forged references is guaranteed at compile time using associated constants in the case trait for unified types. Only remote references can be forged.

2.6 Remote Actors

All remotely accessible actors need to be addressable independently from other actors on the same node. Every actor has a name, which is a combination of a string and a unified type variant. The variant corresponds to the message type the actor receives. When they are started, an actor's names and local reference are sent to a registry, which is itself an actor. When a serialized message is received on a socket, it is forwarded to the registry. The registry resolves the destination name contained in the serialized message to a local reference, and forwards the serialized message if it exists. To improve parallelism, actors are responsible for deserializing their own messages.

2.7 Double Threaded Actors

Most actors run on a single thread, which receives messages, deserializes them if necessary, and runs the Actor trait's recv method. Some actors may want to use 2 threads: a primary task for executing business logic defined by the programmer, and a secondary task for handling other concurrent operations related to the primary. The secondary task will handle these for the primary task:

- **Priority Queuing** We couldn't do this well without a secondary task. The primary task would be unable to build up a queue without pausing in between message processing to rank queued messages. Better to have the highest priority message ready to go when the primary asks for it by delegating to a secondary task. Priority queuing is not yet supported for double threaded actors.

- **Deserialization** If an actor is receiving large, remote messages often enough, deserialization may be taking a significant portion of its compute time. The secondary can use its compute time for deserialization instead, or may spawn a separate task if the serialized form is large enough to warrant doing so (although this is not currently the case).

- **Monitoring** The secondary is be able to track whether the primary panicked without heartbeats. Upon detecting a failure, the secondary will restart the primary in the same state it was in when it crashed, with no cloning of the state. Monitoring is not yet implemented for double-threaded actors.

To process the next message, the primary task sends a request via multiple-producer-single-consumer channels to the secondary, which removes the most important message from its queue and passes it back using another channel.

2.8 Clustering

Perhaps Aurum's most important feature, the clustering module exists to define a group of nodes, who all have knowledge of each other. Knowledge of which nodes are part of the cluster is propagated to other nodes via a gossip protocol, where the data is randomly sent out until every node in the cluster has it. If a node has not received a piece of gossip for a while, it will request one.

Failure Detection

The clustering elastic: nodes can be added or removed from a cluster at any time, and the system will adapt. Should a node fail, it will be noticed through a probabilistic failure detection. Each node keeps a buffer of timestamps on heartbeats it has received from the nodes it is in charge of monitoring. The gaps between these timestamps are arranged in a normal distribution. A probability `phi` (given by the user as a configuration option) will be compared against the cumulative distribution function (CDF) of the time since the last received heartbeat to determine if the node should be considered down. When the time since the last heartbeat reaches a value when the CDF is at or above phi, the node will be marked down. The state change will then be dispersed throughout the cluster with gossip. In the event of an erroneous downing, the node being downed will eventually receive word that is has been downed (through gossip requests). It will assign itself a new identifier and rejoin the cluster.

The Node Ring

Monitoring in the cluster is sharded. Nodes do not track every other node, but a small subset of them. The alternative would lead to a quadratic increase in network traffic as the cluster grew. Instead, nodes are hashed and placed within a consistent hashing ring. The consistent hashing ring has an inverse function, so nodes can see which nodes are in charge of them, and which nodes they have to monitor. The gossip protocol is heavily biased towards sending to a node's neighbors in the ring, because ring states must agree with each other if heartbeats are to work properly. When a node is added, removed or downed (therefore changing the ring), only that node's neighbors in the ring are affected. Using the ring, the entire cluster need not change its monitoring behavior.

Within the ring, each node has a number of virtual nodes (or “vnodes”) representing it at different points in the ring. Vnodes communicate how much relative work a node should be assigned in proportion to the rest of the cluster. A single node having 6 vnodes in the ring while every other node has 3 generally means that node will be performing twice as much cluster work. Because vnodes express a proportion, assiging every node 3 vnodes allots the same amount of relative work to each node as using 1 vnode. Using more vnodes for each node decreases the chances of a node not getting enough work or getting too much work based on a unlucky hash value. Using more vnodes also increases the size of the ring structure, so this is a trade-off. However, because the ring is based on the gossip state, changes to the gossip state causes a node to change its view of the ring. It is never sent over the network, so increased memory use is the only potential problem for a larger ring. Increasing the number of vnodes does not change the size of the gossip state.

Using Cluster

Local actors can subscribe to changes in the state of the local cluster instance. Each update includes the full node ring, a list of active nodes, and a list of events that happened since the last update (adds, removals, downings, and changes to the local node’s identifier). The first update each subscriber gets (no matter when it subscribes) contains the identifier of the local node. Information about a node includes its location, identifier and number of vnodes. Because subscribers have full access to the node ring, they can use it to perform their own sharding. Any hashable value can ask the ring which nodes should be responsible for it.

To join the cluster, a local clustering instance needs to be started. The local instance needs a configuration, number of vnodes for this node, and a list of seed nodes. The list of seed nodes will be continually pinged until the first response, upon which the node is now part of the cluster. If the seed nodes do not response before the limit is reached (defined in the configuration), the node will start a separate cluster consisting of only itself.

Cluster have 2 configuration structs: `ClusterConfig` and `HBRConfig`. `ClusterConfig` contains information that the node will need to interact properly, with the rest of the cluster, including the seed nodes and `replication_factor`, which signifies now many nodes will be managing each node in the cluster. The `replication_factor` must be the same value on every node in the cluster. `HBRConfig` configures the heartbeat receivers for a node's charges, including `phi` values. Each cluster is identified and discovered by a globally known name, which is supplied when starting a cluster instance. To join the same cluster, each instance must start already knowing the name of the cluster they are about to join. A single node can join many cluster as the user wants.

2.9 Delta-State CRDT Dispersal

Conflict Free Replicated Data Types (CRDT), for sharing data between members of a cluster. Updates to these types do not require any coordination, and are eventually consistent. Copies of the data are stored on each member of the cluster, updates are applied via merge functions. Merge functions are a binary operation that takes the old local version and new received version of the data and returns a new local version. Merge operations are commutative, associative, idempotent and monotonic, so CRDTs are useful atop communication protocols with no delivery or message ordering guarantees.

Delta-State CRDTs [20] are optimized structures whose mutation operations return a much smaller version of the state they operated, only containing changes to it. These deltas can be merged with each other just like the full CRDTs they are generated from, sent over the network and merge with the recipient's full state. The result is significantly decreased network traffic, serialization and deserialization time, and retransmission. Each node in the cluster keeps a buffer of deltas representing how far back they have to go in order to make sure the cluster-wide state is consistent, along with acknowledgements informing senders on who has been updated, and at what point to which they have been.

These CRDTs are implemented as persistent data structures like hash array mapped tries, relaxed radix balanced vectors, and b-trees. Rust's `im` (short for "immutable") library [18] thankfully implemented these structures. They can be cloned and disseminated locally at little cost, while having only slightly more expensive mutation operations.

2.10 External Node Tracking

Sometimes, you want to make a node accessible to a cluster without that node joining. In cases where low-latency queries are required, an external node's information and status must be available instantly to every cluster member. The `devices` module provides a way to introduce an external node to a cluster. Among other applications, this is useful for IoT. IoT devices should not be part of the cluster, their variable location makes them unsuitable for highly coordinated interactions.

The servers decide among themselves who will manage each device using the node ring. When the clustering gossip converges, the servers have the same knowledge of the cluster, and each server will know who is supposed to handle each device. Although not all servers are responsible for keeping track of all devices, all servers know of the existence of every device. The devices and their states are stored in a cluster-wide CRDT, which is circulated in the cluster using Aurum's CRDT functionality.

The devices are responsible for initiating contact with the servers. They must be provided with a list of seed servers at startup. It will send its preferred heartbeat interval to every server in its list of seeds. If a server receives an initiation request from a device it is not responsible for, it will forward that request to the server who is. Once a server receives an initiation request from a device, it will start a local actor dedicated to sending heartbeat requests to the device and deciding what the device's state is based on current phi values. When a device receives a heartbeat request from its server, it sends a heartbeat in reply. The device also keeps track of the heartbeat requests it receives from the server using a `phi` accrual detector. If a server failure is detected, the device will contact the seeds again. If multiple senders of heartbeat requests are detected, the device will send a message to every server ordering them to kill themselves if they are not the designated sender.

2.11 The Test Kit

Distributed algorithms cannot make the same assumptions local algorithms can. Any remote message send can fail, be duplicated or delayed by any amount of time. When designing distributed algorithms, the programmer must account for any message having any delivery error. Unreliable delivery is easy to control from an application level without relying on or affecting the rest of the system. The test kit provides macros to select between sending actor messages normally and randomly dropping them. The macros take variables specifying whether the messages should be dropped or not. In modules like cluster and CRDT, these variables will be controlled by compiler flags. The user will be able to configure whether the sends may fail using these flags.

The probability of a message send failure is configured at runtime with instances of `FailureConfig`. Different `FailureConfig` instances are associated with sockets in a `FailureConfigMap`, which has a default `FailureConfig` if one is not found in the map. `FailureConfig`s also contain optional upper and lower bounds for delays in transmission should the random check succeed.

The test kit contains a logging module as well. The logger is an actor, whose reference is stored in the node. Logging levels will be configured per module by compiler flags. There are 7 logging levels (in order of priority): Trace, Debug, Info, Warn, Error, Fatal and Off, and a macro for each level. Each macro accepts a log level as an argument. If the log level passed in is higher that the log level signified by the macro, the log message is not sent. Logging will accept any Display trait object. Currently, the only log target it stdout.

CHAPTER 3

AURUM LIBRARY INTERFACE

3.1 The Actor Trait

The actor trait represents the most basic unit of programming in Aurum. It is a simple interface, with 2 generic types and 3 methods:

```
pub trait Actor<U: Case<S> + UnifiedType, S: SpecificInterface<U>> {
  async fn pre_start(&mut self, ctx: &ActorContext<U, S>) {}
  async fn recv(&mut self, ctx: &ActorContext<U, S>, msg: S);
  async fn post_stop(&mut self, ctx: &ActorContext<U, S>) {}
}
```

`pre_start()` runs before any messages are received. `recv()` is a reaction to a message. `post_stop()` runs after the actor is terminated. Actor requires 2 generic types. `U` is the unified type, the list in which all the message types received by the entire application reside. `S` is the message type received by the actor.

3.2 The AurumInterface Macro

AurumInterface is a derive macro applied to the message types for actors. It generates all the necessary implementations for the message type to interact with the unified type and produce interfaces. Currently, AurumInterface has a single annotation: `aurum`, which has one optional argument, telling AurumInterface whether the interface is to be exclusively local or not. Message types using the AurumInterface macro do not need to know of a specific unified type to belong to. An example:

```
#[derive(AurumInterface)]
#[aurum(local)]
enum MyMsgType {
  #[aurum]
  MyStringInterface(String),
  #[aurum(local)]
  MyNonserializableInterface(&'static str)
  MyOtherMsg(usize)
}
```

In this example, because one of the message options is not serializable, the message type as a whole is not serializable. However, you can use an `ActorRef<MyUnifiedType, String>` to send a string from a remote machine to whatever actor uses this message type. You can also create a `LocalRef<&'static str>`, but not an `ActorRef`.

3.3 The Unify Macro

The `unify!` macro is responsible for constructing the unified type and implementing traits for it. Arguments to `unify!` must include all message types (whether they are to be remotely accessed or not), and types used for remote interfaces. `unify!` creates a type, so it should only be called once in a single application. An example:

```
unify! { MyUnifiedType =
  MyMsgType |
  MyOtherMsgType |
  MsgTypeForSomeThirdPartyLibrary
  ;
  String |
  InterfaceForSomeThirdPartyLibrary
}
```

3.4 Spawning Actors

The node is responsible for spawning actors and managing global system information. It is accessible from every actor it spawns, and contains references to the socket information, Tokio runtime, registry and logger. Creating a node only requires a socket and knowing how many system threads the Tokio runtime will use. Spawning actors is very simple. You need an initial instance of whatever type implements the actor trait, the string part of the actor's name, whether the actor should be double-threaded and whether a reference to it should be sent to the registry.

```
struct MyActor {
  first: String,
  second: usize
}
// Don't forget this annotation, the compiler will whine.
#[async_trait]
impl Actor<MyUnifiedType, MyMsgType> for MyActor {...}

let socket = Socket::new(...);
let node = Node::new(socket, 1);
let actor = MyActor {
  first: "hi there",
  second: 42
};
node.spawn(
  false, // Double-threaded?
  actor,
  "my-very-cool-actor",
  true, // Register?
);
```

3.5 Sending Messages

Using an `ActorRef`, there are several ways to send a message. `remote_send()` uses the remote address of the actor, whether it is local or not. `send()` clones the message if it is being sent locally. `move_to()` takes ownership.

`ActorRef`'s have 3 components: the target host/port, the name of the recipient actor, and a possible `LocalRef`. You may construct the target node and actor name independently. This is just like forging references. The `udp_msg()` function takes these separate components.

```
let socket = Socket::new(...);
let dest = Destination::<MyUnifiedType, String>::new::<MyMsgType>(
  "my-very-cool-actor"
);
let msg: String = "My name is Baloo";
udp_msg(&socket, &dest, &msg);
// Forging a reference
let forged = ActorRef::<MyUnifiedType, String>::new::<MyMsgType>(
  "my-very-cool-actor",
  socket
);
forged.remote_send(&msg);
```

You can send a message unreliably using the `udp_select!` macro. Whether the message is artificially dropped or not depends on a `FailureMode`. `FailureMode` has 3 options: None, Packet (not yet implemented) and Message. `FailureMode::None` will send the message without checking first to artificially drop it. `FailureMode::Message` will atomically decide per message whether the message will be sent, and with what delay.

```
// Doesn't have to be a const
const MODE: FailureMode = FailureMode::Message;
let fail_map = FailureConfigMap { ... };
let msg: String = "My name is Baloo";
udp_select!(MODE, &node, &fail_map, &socket, &dest, &msg);
```

3.6 Logging

Sending messages to the logger is fairly simple. Define a log level for you environment, and call the macro. The logger is accessible from the node, but there is nothing stopping you from spawning your own logger, it's just an actor. The log messages can be anything that implement Display. The argument is turned into a trait object in the macro body.

```
// Doesn't have to be a const
const LEVEL: LogLevel = LogLevel::Debug;
// Not logged, the level is Debug, which is above Trace
trace!(LEVEL, &node, "kelp");
// This is logged, Warn is above Debug
warn!(LEVEL, &node, "sharks");
```

3.7 Starting a Cluster

Each cluster has a string name, which is used to construct the necessary actors. In addition to the configurations, `ClusterConfig` and `HBRConfig`, you need an initial list of subscribers to cluster events (it could be empty), a `FailureConfigMap` and a number of vnodes. Cluster's constructor returns a `LocalRef` that sends commands to the local cluster instance.

```
let mut clr = ClusterConfig::default();
// Changes to clr...
let mut hbr = HBRConfig::default();
// Changes to hbr...
let cluster = Cluster::new(
  &node,
  "my-cool-cluster-name".to_string(),
  3, // number of vnodes
  vec![ctx.local_interface()], // Initial
  fail_map,
  clr,
  hbr,
);
```

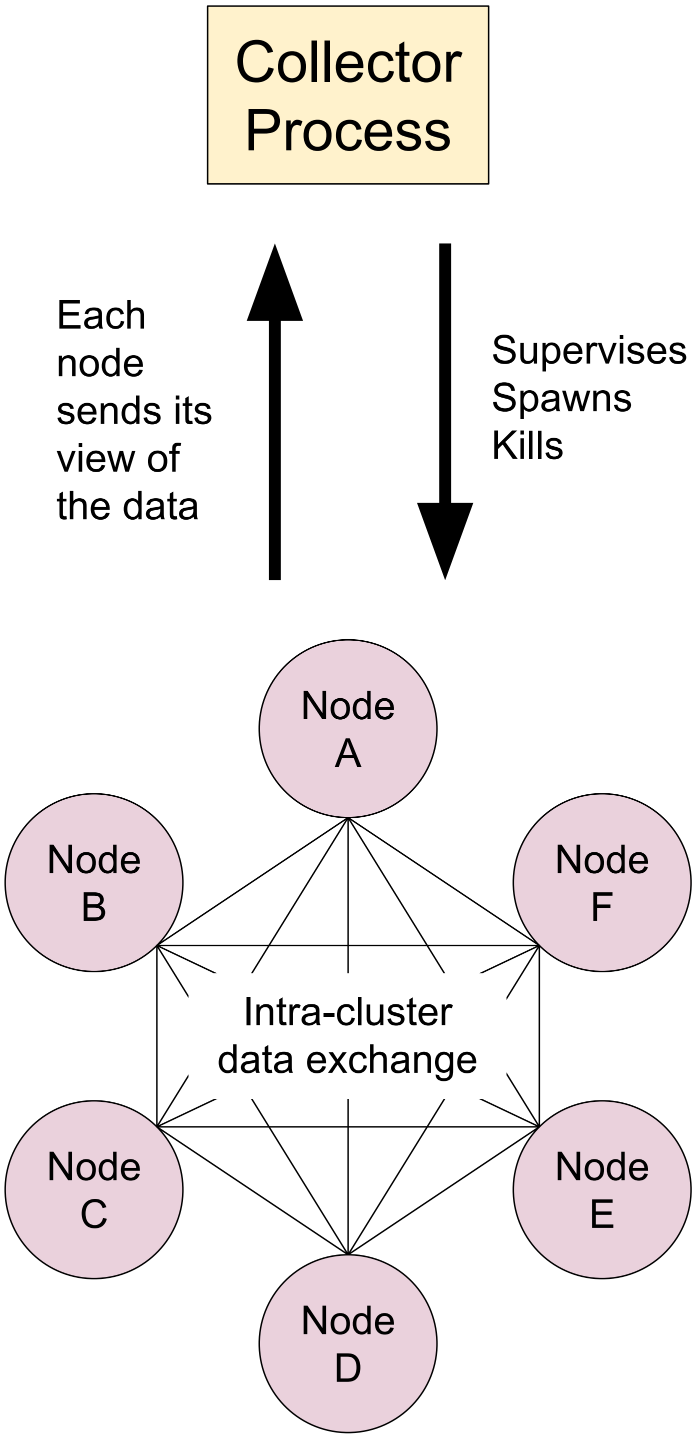
CHAPTER 4

CORRECTNESS TESTING

In addition to the difficulties associated with testing local concurrent code, we must account for the uncertainty of network communication. Custom data structures like the device CRDT, the gossip state and the node ring are unit tested. Actors involved in complex protocols are too tightly coupled together for unit testing to be effective. Instead, actors are tested from a user's perspective. Many features (like clustering) involve actors interacting with other instances of themselves on different nodes. Tracking how cluster-wide state evolves over time is best done from an application level.

Our tests for clustering, device tracking and CRDT dispersal make use of the testkit to randomly inject failures into message sends. These tests have a single central node spawn many processes that cluster with each other. The central process subscribes to updates from every node, and executes a series of events (like node spawns, terminations and data mutations). For each event, the central process constructs an expected value for the shared data, and waits for that value to be received from every process before proceeding to the next set of events. This allows the test to be conducted without use of precise timeouts for every message. It provides the flexibility we need to use the same test configuration for many tests with different failure values and delays. Figure 4.1 provides a visual aid.

Figure 4.1: Test Architecture for Distributed Algorithms



CHAPTER 5

PERFORMANCE COMPARISONS

5.1 Introduction

We will compare the performance of Aurum to Akka. We need a benchmark to show the performance difference. We will discuss existing benchmarks and why they do not fit our use case, then define our own benchmark to compare performance.

5.2 Existing Benchmarks

Many benchmarks exist for some types of distributed software, including distributed stream processing systems (Apache Spark [24], Apache Storm [25] or Apache Flink [26]), big data engines (Apache Hadoop [27], Apache Spark or MapReduce) and actor models (Akka, Proto.Actor or Erlang). Although performance of these systems is effectively measured, effects of failure on performance are not explored.

MRBench [21] is a benchmark created for MapReduce. It implements various SQL features in MapReduce (such as JOIN and WHERE), and ran a variety of complex queries against gigabyte-size databases stored in the Hadoop Distributed File System. Since their benchmark operates on static data, it does not fit an IoT use case, which requires data processing to be in real time.

RIoTBench [22] is a collection of micro-benchmarks that can be strung together in task graphs to produce fairly realistic test applications. It uses Apache Storm to execute these tasks, which range from transformation and filtering to prediction. RIoTBench and other similar suites are meant to test distributed stream processing systems, which perform SIMD operations on potentially real-time streams of data. In these streams, data exclusively travels one-way: from the devices to the cluster. More complex IoT applications require continuous, full-duplex interaction between devices and the cluster.

Although distributed stream processing systems are designed to tolerate failures in the cluster, the benchmarks do not test systems under this kind of strain. Effects of failure on performance are not well established by these benchmarks. Cardoso et al [23] used a simple ring benchmark to explore the performance of Erlang and Scala actors, but no clustering logic was present in the test (no capacity for adding/removing nodes or failure detection. Because failure is relatively common in IoT settings, understanding impacts of device failures is especially valuable. Performant fault tolerance depends on both the recovery time of the device and its process, as well as the speed at which the cluster handles its membership. A benchmark containing full-duplex device interaction, fault tolerance and clustering logic will be developed to address these requirements.

5.3 Our Benchmark

The benchmark will test the `devices` module against its Akka equivalent. It will consist of a group of external nodes periodically sending data to a cluster in a data center. Each time a node sends data to the cluster is a “report”. Reports consist of data aggregated between the last report and the time it is sent. The cluster request a report from every external node, and upon receiving it, will make another request right after. This allows each server to set its own pace without being overloaded with requests.

Failures are expected to occur often in this environment. Broadly speaking, failures come in 2 categories: hardware and software. Hardware failures result in a full restart for the devices, software failures restart the daemon in which our device node is running. Similar failures can happen within the cluster.

MTTR (Mean Time To Repair) can be used to measure how quickly devices recover from failure. "Repair" is defined as the recovered device's status is now up to date and known to all members of the cluster. MTTR is useful for testing how quickly the system can recover.

We want to know how many reports per second (throughput) can be processed end-to-end under different conditions. MTTF (Mean Time To Failure) is the average frequency of failures injected into a running system to test its fault tolerance. We will test what throughputs can be achieved with different MTTF values. Injected failures can be hardware or software failures, in either the cluster or devices. Which types of failures MTTF values affect will be reported alongside throughput. We will start with very high MTTF values (i.e. infrequent failure), and decrease until the system can no longer function. We aim to show that Aurum remains viable and continues to work correctly, albeit at slower throughput.

The Benchmark's Parameters

To simulate a real IoT environment, it is possible to inject failures and delays into message sends. The probability of a send failing is configurable, along with minimum and maximum delays for sending a message. Processes can be killed, then restarted automatically at randomly generated intervals in a specified range. The devices all use the same heartbeat interval (although the protocol allows them not to), which is configurable.

The Business Logic

In each server, the business logic contacts the middleware and subscribes to the list of devices that server is responsible for. For each device, the server will send report requests to the device. The device responds with a report, simply an array of 64-bit signed integers numbered 1 to 1000. If a report is not received for a while, the request is resent. When reports arrive at the server, the numbers are summed, then added to a total, and another report request is sent. Each device on each server has its own total. The collector (a single process) will make periodic requests to each server, asking for its totals for each device. The collector distills the totals into a number of reports received by the cluster as a whole. This information will be used to tell how many reports the cluster is able to process as a whole.

5.4 Results

Our benchmark was run on Boise State University's Onyx cluster. Nodes that run client processes usually run more than one.

Aurum versus Akka

Our first set of tests compares our library to Akka. The algorithm running on servers was designed not to break on higher loads, but merely slow down throughput. The effect can be seen when the throughput plateaus even as the load gets much higher than it did when maximum throughput was achieved. These tests were performed with no failure injection.

The first series of tests we ran used 1 server and 3 client nodes, increasing the number of clients per node with each test. Akka's throughput plateaued rather quickly at 14,000 reports per second, but stayed constant as the number of clients increased. Aurum took much longer to plateau, but eventually did at 42,000. Results are in Table 5.1 and Figure 5.1.

The second series is exactly the same, but scaled up to 11 servers and 18 client nodes. This provides 21 clients on average to each server, allowing Akka to plateau. In accordance with the limit we saw in the previous test, Akka's throughput plateaued at 155,000 reports per second. Aurum's throughput did not stop increasing as the number of clients went up, going all the way to 315,000 reports per second. Results are in Table 5.2 and Figure 5.2.

Aurum is able to consistently outperform Akka. The higher the load, the more pronounced this difference becomes. For a single server, Aurum's peak throughput is triple that of Akka's.

Dropping Messages

To simulate a realistic IoT application, we tested Aurum with messages randomly dropped instead of sent, with a particular probability. Messages are only dropped when sent between clients and servers, not between servers. To keep the loads high, no delays were injected. If a report is not received by the server when requested, the server will resend the request after a 3 millisecond timeout. This test used 11 servers and 18 client nodes with 13 clients per node. We gradually increased the probability of a message failing to send, and tracked throughput alongside it. As the probability increased, we would expect the throughput to approach zero in a monotonically decreasing, concave up fashion. Results in Table 5.3 and Figure 5.3 show our expectation to be correct. Aurum remains usable (although slower) on higher rates of message drops.

Failing Clients

A realistic IoT application may include regular client failures. To simulate that, each client node has a supervisor process that kill and restarts clients as the test runs, with a certain MTTF. The time between starting a process and killing it is randomly generated between MTTF*0.25 and MTTF*1.75. A client is always restarted 5 seconds after it is killed, regardless of how long it was alive. As MTTF decrease, we expect the throughput to approach zero, monotonically decreasing and concave down. Results in Table 5.4 and Figure 5.4 show the curve monotonically decreasing and is not perfectly concave down, but that is the general trend.

Failing Servers

Although servers failing regularly is not part of a realistic application, we ran the benchmark for the sake of completeness. This test has exactly the same setup as the client failures. The external node tracking protocol is not designed for such high failure rates, so we expect inconsistencies for higher MTTF values. Results in Table 5.5 and Figure 5.5 show the throughput to be non-monotonic, with a small spike at a MTTF of 21 seconds. All clients use a comprehensive list of seed nodes, so multiple servers may be receiving from a client some of the time. This increase is counteracted by the throughput loss caused by server processes being killed, which gives us the inconsistencies at higher values. As values get lower, process deaths become the dominant factor and throughput starts approaching zero in the same way as the client failure test.

Analysis

The performance gap between Aurum and Akka is not fully understood. However, we can conjecture possible contributors to the difference. Tokio's scheduler is co-operative, and is explicitly notified by the code whenever `await` is called, which potentially speeds up context switching. Reflection is implemented efficiently in Aurum. Enum variants are represented with a small number, and the dispatch to the proper deserialization function can be performed quickly. Being a JVM framework, Akka dynamically allocates more memory than Aurum and must free that memory with garbage collection. The significance of these factors could be captured by future experiments. Akka uses Scala's ExecutionContext, which could be compared to Tokio. Reflection is more difficult to compare, but tests are possible when it is combined with serialization. Garbage collection is difficult to control, so the exact impact of increased dynamic memory allocation on performance will be challenging to test.

# clients/node	Aurum reports/second	Akka reports/second
1	5,500	4,300
2	11,000	8,500
3	15,500	11,000
4	18,500	13,000
5	21,500	14,000
6	25,000	14,000
7	28,500	14,000
8	33,000	14,000
9	37,000	14,000
10	40,500	14,000
11	41,500	14,000
12	42,000	14,000
13	42,000	14,000

Table 5.1: Onyx Cluster - 1 Server - 3 Client Nodes - No Failures

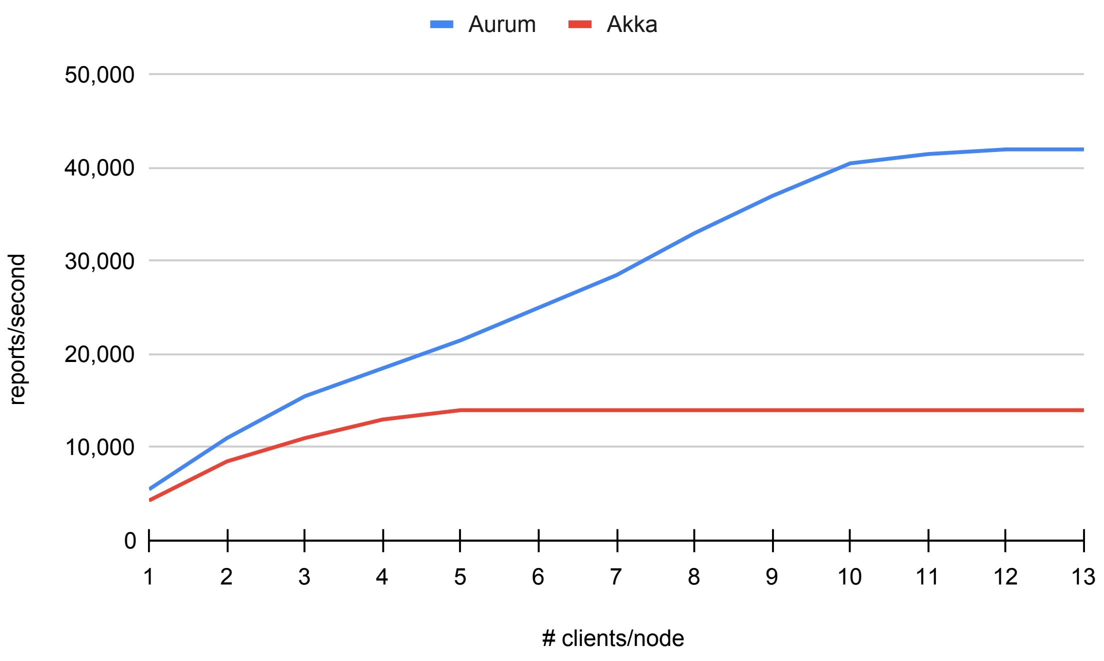


Figure 5.1: Onyx Cluster - 1 Server - 3 Client Nodes - No Failures

# clients/node	Aurum reports/second	Akka reports/second
1	30,000	25,000
2	70,000	50,000
3	105,000	70,000
4	130,000	95,000
5	155,000	110,000
6	170,000	135,000
7	190,000	145,000
8	210,000	155,000
9	235,000	155,000
10	255,000	155,000
11	275,000	155,000
12	300,000	155,000
13	315,000	155,000

Table 5.2: Onyx Cluster - 11 Servers - 18 Client Nodes - No Failures

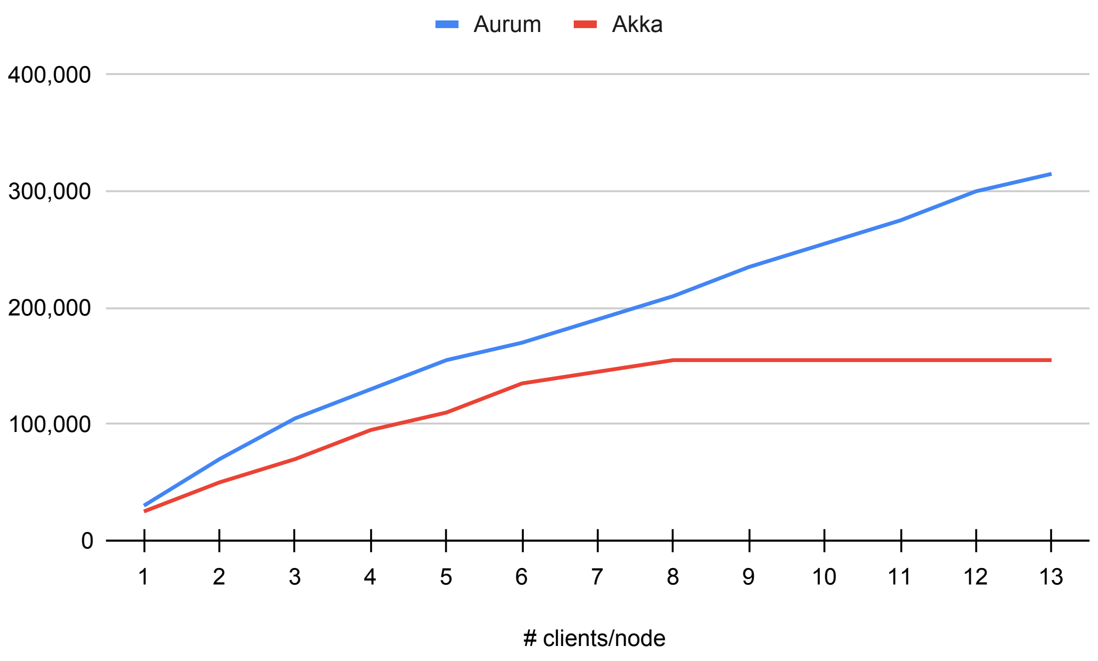


Figure 5.2: Onyx Cluster - 11 Servers - 18 Client Nodes - No Failures

drop %	reports/second
0%	315,000
5%	215,000
10%	150,000
15%	105,000
20%	80,000
25%	60,000
30%	45,000
35%	36,000
40%	28,000
45%	21,000
50%	16,000

Table 5.3: Onyx Cluster - Message Drops

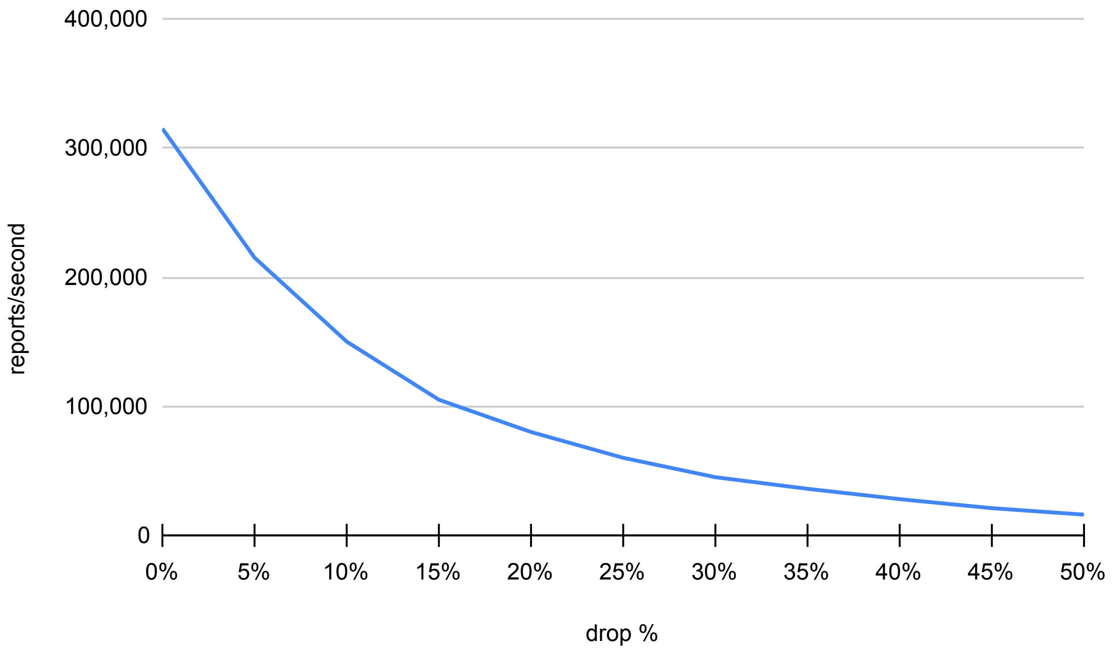


Figure 5.3: Onyx Cluster - Message Drops

Mean Time to Failure (s)	reports/second
30	277,000
27	274,000
24	269,000
21	265,000
18	238,000
15	232,000
12	223,000
9	204,000
6	179,000
3	120,000
2	103,000
1	66,000

Table 5.4: Onyx Cluster - Client Failures

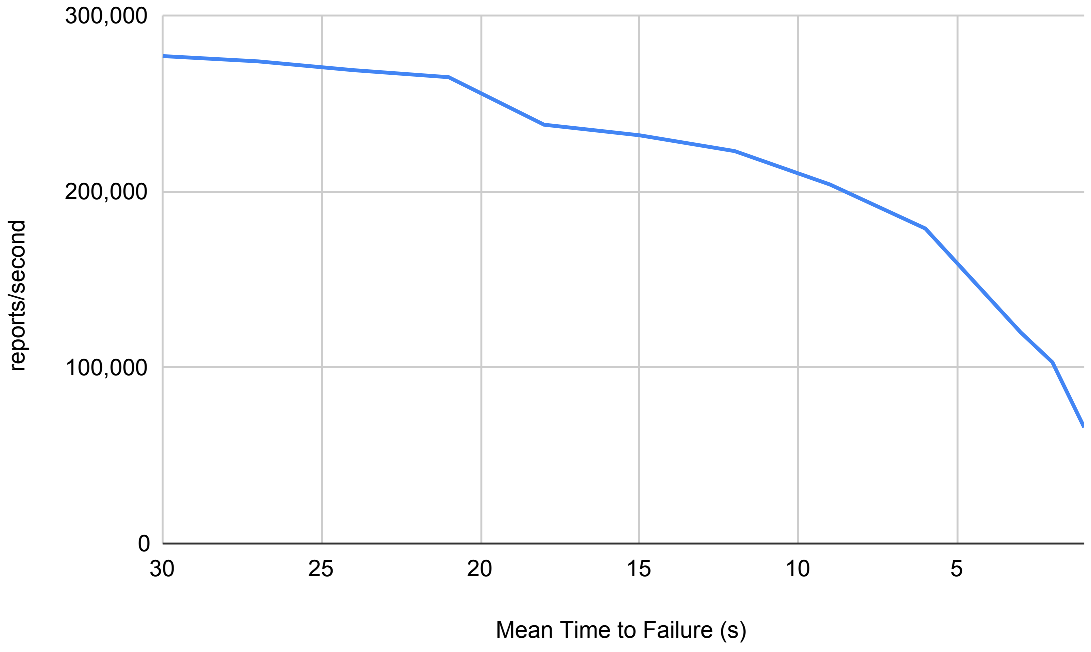


Figure 5.4: Onyx Cluster - Client Failures

Mean Time to Failure (s)	reports/second
30	240,000
27	235,000
24	210,000
21	230,000
18	217,000
15	200,000
12	170,000
9	155,000
6	110,000
3	47,000
2	33,000
1	6,000

Table 5.5: Onyx Cluster - Server Failures

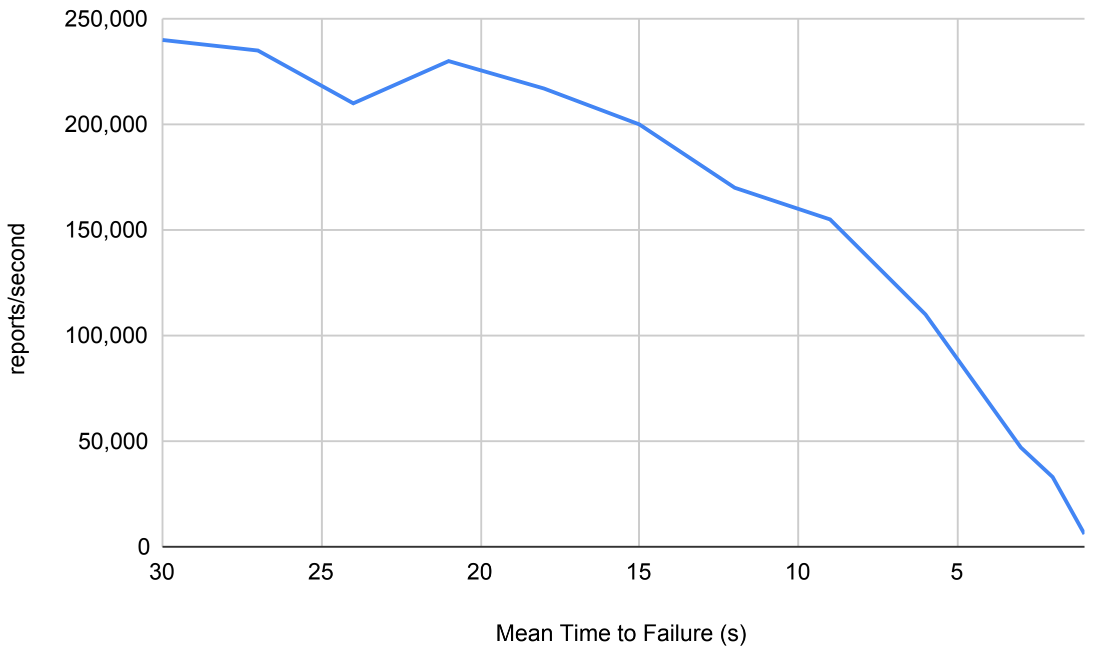


Figure 5.5: Onyx Cluster - Server Failures

CHAPTER 6

CONCLUSION

6.1 Summary

Aurum is a type-safe and performant library. It outperforms Akka in our benchmarks. The performance benefits of using Aurum become more apparent as we scale up. An Aurum server can deal with dozens of clients at full throttle. Like Akka, Aurum is a fault tolerant system, and continues to function properly even as processes crash and messages are dropped. Aurum's interface is simple and powerful. It provides easy ways to forge references, configure and launch clusters, and inject message drops and delays into every part the application. IoT applications are easy and safer to write in Aurum, thanks to its forgeable references, message type translations, and extensive compile-time checks.

6.2 Future Work

Although a good start has been made to the Aurum library, there are more features that must be implemented for it to be fully usable in production. Below we describe some of these features.

TCP and TLS Support

Aurum communicates exclusively through UDP at this point in development. Users may want reliable delivery with TCP and security through TLS [17]. Actor messages are fire-and-forget, so the sender should not be the one keeping track of TCP connections. The node will manage actors who own TCP connections.

Encryption for UDP Messages

TLS needs a reliable, in-order transport protocol like TCP underneath it to function properly, so it cannot be used over UDP. However, secure UDP communication would be incredibly useful to have. The proper interface for this feature is unclear. Most complexity lies in managing keys. It may be beneficial to use a single, cluster-wide key that is shared using asymmetric encryption, or have every node with its own key. *Distributed Systems* [29] discusses ways to manage keys. The keys will also need to be easily accessible by every actor on a node at all times.

More Options for Downing

The clustering module only has one mechanism for deciding if a node is down. If a single monitor detects failure, that detection will be propogated to other nodes, who will also consider the node down. This does not effectively deal with network partitions. Network partitions may be especially problematic for features that guarantee only a single, cluster-wide instance of a particular entity exists. Virtual actors and distributed mutexes are examples.

Fully Featurize Double Threaded Actors

Priority queuing will require several additional traits. Actor message types will need a new trait with a method returning that message's priority. Another trait will be needed to provide a generic way for the programmer to structure priority queuing. Monitoring can naturally follow from the features in Tokio's message passing. When the secondary detects the crash through the message sender, it will restart the primary. The primary's state may be owned by a mutex, to enable mutable access on data shared between the primary and the secondary. Restoration after a crash will simply involve cloning a reference to the mutex and passing it to the newly started primary.

Expand the CRDT Module

There are many CRDTs not yet added to the module, including grow-only sets, counters, observe-remove maps and sets, registers and much more. Direct support of named delta-state CRDTs need to be added to the interface as well.

Heavy Compute Operations

Aurum is backed by the asynchronous runtime, Tokio. Rust's asynchronous features are meant specifically for IO-bound tasks. Spawning CPU-bound tasks on an asynchronous runtime will lead to starvation. By adding second pool to the node that schedules CPU-bound tasks, Aurum can handle a much greater variety of applications. One such pool is found in the Rayon library [19], which has a convenient interface for parallel computing. Rayon's thread pool is independently managed from Tokio's, but results from compute operations executed with Rayon can be passed to an actor asynchronously without interfering with Tokio's scheduling of other actors.

Virtual Actors

It may be useful to only refer to an actor by its logical name, rather than its physical location. If an actor's host machine crashes, it should be restarted on another node in the cluster. Messages sent to that actor are rerouted accordingly. Such actors are called virtual actors, and other actor libraries like Akka and Orleans have already implemented them. Virtual actors have the potential give an application better programmability and resilience.

Investigate Performance Differences

We would like to run more experiments in the future, to more fully understand why Aurum achieves better performance. The nature of our benchmarks may change as Aurum's feature set expands and its performance further improves.

REFERENCES

[1] Hewitt, C. (2015, January 21). Actor Model of Computation: Scalable Robust Information Systems. Retrieved December 03, 2020, from `https://arxiv.org/abs/1008.1459`

[2] Sánchez, D. D., Sherratt, R. S., Arias, P., Almenarez, F., Marín, A. (2015, June 24). Enabling actor model for crowd sensing and IoT. Retrieved December 2, 2020, from `http://centaur.reading.ac.uk/43187/1/Enabling%20Actor%20Model%20for%20Crowd%20Sensing%20and%20IoT%20Full%20text.pdf`

[3] Erlang Reference Manual User's Guide. (2020, September 22). Retrieved December 02, 2020, from `https://erlang.org/doc/reference_manual/users_guide.html`

[4] Riker-Rs. (2020, November 30). Riker-Rs. Retrieved December 02, 2020, from `https://github.com/riker-rs/riker/`

[5] Bastion. (2020, November 6). Retrieved December 02, 2020, from `https://github.com/bastion-rs/bastion`

[6] Bonet, D. (2020, November 29). Acteur. Retrieved December 03, 2020, from `https://github.com/DavidBM/acteur-rs`

[7] Actix. (2020, November 20). Retrieved December 03, 2020, from `https://github.com/actix/actix`

[8] Akka Documentation. (2020). Retrieved December 03, 2020, from `https://doc.akka.io/docs/akka/current/index.html`

[9] Akka Serialization Between Local Actors. (2020). Retrieved December 03, 2020, from `https://doc.akka.io/docs/akka/current/serialization.html#external-akka-serializers`

[10] The Rust Reference. (2020). Retrieved December 03, 2020, from `https://doc.rust-lang.org/stable/reference/`

[11] Dart Isolate. (n.d.). Retrieved December 03, 2020, from `https://api.dart.dev/stable/2.10.4/dart-isolate/Isolate-class.html`

[12] Barney, B. (2020, December 2). Message Passing Interface. Retrieved December 03, 2020, from `https://computing.llnl.gov/tutorials/mpi/`

[13] Kubernetes. (2020). Retrieved December 07, 2020, from `https://kubernetes.io/`

[14] Futures RS. (2020, December 2). Retrieved December 07, 2020, from `https://github.com/rust-lang/futures-rs`

[15] Tokio. (2020, December 7). Retrieved December 07, 2020, from `https://github.com/tokio-rs/tokio`

[16] Serde. (2020, December 5). Retrieved December 07, 2020, from `https://github.com/serde-rs/serde`

[17] Rescorla, E. (2018, July). The Transport Layer Security (TLS) Protocol Version 1.3. Retrieved May 07, 2021, from `https://tools.ietf.org/html/rfc8446`

[18] Stokke, B. (n.d.). Bodil/im-rs. Retrieved May 07, 2021, from `https://github.com/bodil/im-rs`

[19] Stone, J. (n.d.). Rayon-rs/rayon. Retrieved May 07, 2021, from `https://github.com/rayon-rs/rayon`

[20] Almeida, P. S., Shoker, A., & Baquero, C. (2018). Delta state replicated data types. Journal of Parallel and Distributed Computing, 111, 162-173. doi:10.1016/j.jpdc.2017.08.003

[21] Kim, K., Jeon, K., Han, H., Kim, S., Jung, H., & Yeom, H. Y. (2008). MR-Bench: A benchmark for mapreduce framework. 2008 14th IEEE International Conference on Parallel and Distributed Systems. doi:10.1109/icpads.2008.70

[22] Shukla, A., Chaturvedi, S., & Simmhan, Y. (2017). RIoTBench: An IoT benchmark for distributed stream processing systems. Concurrency and Computation: Practice and Experience, 29(21). doi:10.1002/cpe.4257

[23] Cardoso, R. C., Hübner, J. F., & Bordini, R. H. (2013). Benchmarking communication in actor- and Agent-based Languages. Engineering Multi-Agent Systems, 58-77. doi:10.1007/978-3-642-45343-4_4

[24] Apache Spark™ - unified analytics engine for big data. (n.d.). Retrieved May 07, 2021, from `https://spark.apache.org/`

[25] Apache Storm. (n.d.). Retrieved May 07, 2021, from `https://storm.apache.org/`

[26] Apache Flink: Stateful computations over data streams. (n.d.). Retrieved May 07, 2021, from https://flink.apache.org/

[27] Apache Hadoop. (n.d.). Retrieved May 07, 2021, from https://hadoop.apache.org/

[28] Cardoso, R. C., Hübner, J. F., & Bordini, R. H. (2013). Benchmarking communication in actor- and Agent-based Languages. Engineering Multi-Agent Systems, 58-77. doi:10.1007/978-3-642-45343-4_4

[29] Steen, M. van, & Tanenbaum, A. S. (2017). Distributed systems. Pearson Education.

APPENDIX A

APPENDIX A: SOURCE CODE

The complete source code can be found here: `https://github.com/arjunlalshukla/research/tree/master/thesis`. Akka's version of the benchmark is found in the `akka-benchmark` directory. Aurum's source code is in the `aurum` directory. The projects are structured as standard `sbt` and `cargo` projects, respectively. Instructions for how to use the code are in the repository.

APPENDIX B

APPENDIX B: TEST ENVIRONMENT SPECIFICATIONS

Language versions:

- Java version: 11.0.11
- Scala version: 2.13.1
- Rust version: 1.51

Onyx `/etc/os-release`:

```
NAME="Red Hat Enterprise Linux Server"
VERSION="7.9 (Maipo)"
ID="rhel"
ID_LIKE="fedora"
VARIANT="Server"
VARIANT_ID="server"
VERSION_ID="7.9"
PRETTY_NAME="Red Hat Enterprise Linux"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:redhat:enterprise_linux:7.9:GA:server"
HOME_URL="https://www.redhat.com/"
BUG_REPORT_URL="https://bugzilla.redhat.com/"

REDHAT_BUGZILLA_PRODUCT="Red Hat Enterprise Linux 7"
```

```
REDHAT_BUGZILLA_PRODUCT_VERSION=7.9
REDHAT_SUPPORT_PRODUCT="Red Hat Enterprise Linux"
REDHAT_SUPPORT_PRODUCT_VERSION="7.9"
```

Onyx head node `lscpu`:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                48
On-line CPU(s) list:   0-47
Thread(s) per core:    2
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz
Stepping:              4
CPU MHz:               919.006
CPU max MHz:           3000.0000
CPU min MHz:           800.0000
BogoMIPS:              4200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              16896K
NUMA node0 CPU(s):
   0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46
```

```
NUMA node1 CPU(s):
   1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep
   mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss
    ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
   arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
   aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx
   est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2
   x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
   lahf_lm abm 3dnowprefetch epb cat_l3 cdp_l3 invpcid_single
   intel_pt ssbd mba ibrs ibpb stibp tpr_shadow vnmi flexpriority
   ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid
    rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt
   clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 cqm_llc
   cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts
    pku ospke md_clear spec_ctrl intel_stibp flush_l1d
```

Onyx nodes `lscpu`:

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              6
On-line CPU(s) list: 0-5
Thread(s) per core:  1
Core(s) per socket:  6
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
```

```
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i5-8500T CPU @ 2.10GHz
Stepping:            10
CPU MHz:             3402.976
CPU max MHz:         3500.0000
CPU min MHz:         800.0000
BogoMIPS:            4224.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            9216K
NUMA node0 CPU(s):   0-5
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep
   mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss
    ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
   arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid
   aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx
    smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2
   x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
   lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd
    ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad
   fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx
   rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1
   xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window
   hwp_epp md_clear flush_l1d
```