Abstract—A distributed system typically has a set of software components located on different computing machines connected over a network. These components communicate through message passing. In this paper, we are interested in studying the key properties of Distributed Computing by building a prototype of an Actor system using Scala/Akka programming language. Specifically, this paper demonstrates how the actor model can be used in implementing distributed computing across connected heterogeneous computing machines. To achieve this objective, we built a distributed runtime environment to support the deployment and execution of distributed actor-based applications on different computing machines. The distributed run-time environment consists of connected runtime environments, which support the execution of individual application components (actors) and managing their communication. The programmability benefits of our runtime environment are evaluated by developing two distributed applications: (1) A peer-to-peer chat application between two heterogeneous machines over a wireless network implemented using Akka; and (2) A computational-intensive application across three connected heterogeneous devices. Finally, we evaluated the performance of the developed prototype experimentally using different metrics.

Keywords—Distributed Computing; Heterogeneous Devices; Scala; Akka; Actors.

I. INTRODUCTION

It is becoming increasingly important to support the deployment and execution of parallel and distributed systems across heterogeneous machines. Consider, for example, an IoT system which involves a large number of end-devices (e.g., sensors and actuators) connected to gateways, which act as the aggregation points for a group of sensors and actuators to coordinate the connectivity of these devices to each other and to an external network. In many cases, both end-devices and gateways operate in resource-constrained concurrent environments with limited memory and power, and require real-time capabilities in some scenarios such as in the Agricultural field [1]. To accommodate such requirements, a distributed run-time environment is needed to be in place to manage the entire system efficiently.

One way to support the functional needs of such systems is by implementing them using Actors [2]. The Actor model is a well-established model for formalizing object-oriented concurrent computations in open and distributed systems. There is a growing number of implementations of Actors, including production languages such as Scala [3] which supports actors through its Akka library [4]. Akka can be used to build highly concurrent, distributed and fault-tolerant applications which can span across multiple processor cores and networks. Akka provides a high-level of abstraction to low-level synchronization mechanisms of multi-threaded applications by hiding them from programmers, allowing them to focus on application-specific details.

The programming required for developing a new concurrent and distributed application across heterogeneous machines can be significant if done from scratch. However, having a runtime environment running on these machines, which can support the deployment and execution of actors, could simplify the burden of developing such class of applications. In this paper, we present a runtime environment will offer high-level primitives supported by a middleware implementing fundamental mechanisms required by distributed actor-based applications. Our solution to such support is implemented for Scala/Akka programming language. The paper also presents two prototype implementations of qualitatively different applications, which are developed on the top of the runtime environment, to illustrate the ease with which new applications can be programmed.

The rest of the paper is organized as follows: Section II presents related work. Sections III and IV present the design and prototype implementation of the runtime environment we have developed for Akka. Section V experimentally establishes the performance cost of executing a large number of actors on heterogeneous machines. Finally, Section VI concludes the paper.

II. RELATED WORK

There has been growing interest in supporting distributed computing from various perspectives, focusing on concerns from communication and coordination...
mechanisms (e.g., [5]), to customized operating systems (e.g., [6]–[9]), to runtime environments (e.g., [10]–[12]), to resource coordination (e.g., [13], [14]). We discuss some of these approaches below.

The programming required for offering a new distributed application can be significant if done from scratch. However, there is an opportunity created by the similarity in the patterns of communication required for distributed applications, especially for IoT applications where contextual data offered by a number of contributors becomes the basis for the application. This pattern of communication was originally defined in [5] as multi-origin communication. Multi-origin communication can be used to support the complex communication and coordination requirements of distributed applications in different domains. For instance, multi-origin communications have been identified as a key mechanism underlying crowd-sourced services [5], and then implemented in the form of a distributed middleware, CSSWare [15]. CSSWare allows a rich variety of crowd-sourced services to be implemented with relative ease, from crowd-sourced recommendation services for restaurants to social media services similar to Twitter.

To accommodate the limited-resource requirements of heterogeneous resource-limited devices such as IoT end-devices, small operating systems, called IoT OSs, are specially designed to manage the device’s resources efficiently. Over the last decade, several OSs for the IoT have emerged such as TinyOS [6], FreeRTOS [7], RIOT [8] and Android Things [9].

The runtime environment of distributed resource-limited applications, which runs over the OS, is responsible for providing all necessary services and support for the execution of applications. There are a number of related works in the literature which provide such support such as PatRICIA [11], RapIoT [12], and Eclipse Kura [16]. These platforms are briefly discussed below.

There have been a few attempts to deploy actors in mobile programming platforms such as in (DAMMP) [17]. A mobile extension to the actor-based Distributed Selector (DS) programming model was proposed to enable programmers to react and adapt to dynamic changes in the mobile device availability. The Android-based platform was evaluated using some mobile devices in a distributed and dynamic setting. Also, the proposed model was set out to reduce issues related to mobile grid computing such as power consumption and heat dissipation. Charousset et al. [18] proposed an actor extension for C++ called native actors in the form of a scalable software platform for distributed and heterogeneous environments. The proposed software platform tried to use the in-design scalability characteristics to support the development of scalable and efficient parallel software in heterogeneous hosting environments.

The programming of resource-efficient concurrent applications is discussed in [19] where a C++ Actor framework, CAF, is introduced which aims to provide a scalable environment for building resource-efficient applications and distributed systems based on the Actor Model. CAF provides a way for programmers to monitor the performance of their distributed systems on the runtime through interactive shells, which gives insights about the runtime characteristics of them. CAF also grants programmers convenient access to aggregated information about resource usage on each node where distributed systems are deployed.

There is some existing work which targeted the challenge of implementing resource control library for Akka [13], [14], which allows cloud services implemented using Akka to support multiple clients from the same instance of their system, with the ability to offer differentiated qualities of service.

III. DESIGN

We took a multi-agent systems approach to support the programmability of distributed applications, broadly founded in the Actors model of concurrency [2]. Where Actors offer significant advantages for implementing large-scale distributed systems, they also offer a rich and clean reference model for research in concurrent systems [20].

In this paper, we present an actor-based runtime environment to support the deployment and execution of distributed applications on heterogeneous computing machines including IoT end-devices. The runtime environment enables programmers to write actor programs which can run on a large number of devices. Actors act the building blocks for an application which will be deployed on the top the runtime environment. The actors are connected in a dataflow to form the application. This simplifies actor migration between runtimes and matching of actor requirements with runtime capabilities. If the runtime does not meet the requirements posed by a currently deployed actor, then the actor will be automatically migrated to a runtime that can satisfy the requirements.
Figure 1 illustrates the software stack of a computing device in our system which includes the following layers: (i) Hardware Abstraction Layer (HAL) is a software layer that enables access to the hardware components of computing devices such as RAM, ROM, serial interfaces, etc. (ii) OS Layer: Computing devices have either embedded or Real-Time Operating Systems (RTOS) that are particularly suited for small constrained devices such as IoT end-devices, and that can provide communication-specific capabilities. (iii) Runtime Environment Layer: The actor-based runtime environment runs over the OS layer. It hosts several actors which can interact with each other both locally and over the network. The runtime provides a uniform computing environment for all actors, regardless of hardware or operating system differences. (iv) Application Layer: Actor-based applications are deployed on top of the runtime environment. Applications are defined as a set of functions which are represented as actors, where each actor can provide data generation (e.g., sensors), processing and data consumption (e.g., actuators). Actor code execution is triggered by certain events such as receiving a new message from another actor locally or remotely, or detecting a change in the state of a sensor.

As shown in Figure 2, the system architecture consists of connected runtime environments running on heterogeneous computing devices, which support the execution of actors and managing their communication. We added a gatekeeper component to decide whether to deliver or postpone the delivery of messages to an actor according to some priority settings set by the programmer. When a sender actor sends a message—through its local runtime environment— to another actor hosted in a remote runtime environment, the gatekeeper dispatches the inbound message to the message dispatcher component, which in turn transports the message to the remote receiving actor using the routing actor.

IV. IMPLEMENTATION

We built a set of programming constructs implemented in a runtime environment to support the programmability of actor-based distributed applications. We prototyped the distributed runtime environment as an actor system. Our implementation is built using Scala programming language which supports actors through its Akka library.

In Akka, a message dispatcher is considered the core engine for the runtime system because it controls the processor cycles given to actors. The dispatcher has access to the global message queue, actors’ mailboxes, and the pool of threads which executes the actors. One of the necessary configuration settings to Akka message dispatcher is throughput, which defines the number of messages delivered to an actor at one time. For example, if the throughput is set to m, and the number of messages queued up in the global message queue for an actor is n, if m < n, m messages are delivered to the actor in one shot, and the remaining messages wait for the next turn.

Figure 3 illustrates the life cycle of message dispatching in Akka. When a message is sent to an actor, the dispatcher first places it in the global message queue. When it is that actor’s turn to be executed, its mailbox is wrapped into an idle thread from the thread pool to create a Runnable Thread. The dispatcher then moves the right number of messages for that actor— as determined by the throughput setting— from the global queue to the actor’s mailbox. Once a thread has been given messages for an actor to execute, for each message (beginning with the top message in the queue), the gatekeeper examines the priorities for executing it, and finally tells the runnable to execute the actor for those messages only if permissible.

We evaluated the programmability benefits of our runtime environment by: (1) Implementing a peer-to-

---

2Mailbox is the dispatching unit in Akka, which contains one or more messages that can be processed in sequence during an interval.
peer chat application between two heterogeneous machines over a wireless network; and (2) Implementing a computationally intensive application across three connected heterogeneous machines. Below, we present the prototype implementations of these two qualitatively different applications to illustrate the ease with which new applications can be programmed.

A. Chat Application

Consider a realtime peer-to-peer chat application between two heterogeneous devices communicating using asynchronous actor messages. We implemented such an application on the top of our runtime environment using the remoting capability of the actor system. We made use of a laptop and Raspberry Pi 3 device to test the developed application. The chat is started by an actor --which is hosted on an instance of our runtime environment-- at the laptop sending a message to another actor hosted at the Raspberry Pi.

First, the local runtime environment has to open an inbound/outbound TCP connection on port 2554. Then, it sends a handshake message to the remote runtime is listening for incoming association. Once the inbound association is accepted at the remote runtime, messages between the two devices start to flow.

B. Word Counter Application

We implemented a distributed word count application that calculates a running word count from a continuous stream of sentences. This application involves many of the structures, techniques, and patterns required for more complex computation which can be supported by our runtime environment. We used the master-slave architecture to show how our runtime environment can be used to break a computationally intensive task down into small subtasks for individual distributed actors to handle. This application demonstrates the coordination and routing capabilities of our system. Remote coordination is the ability for a local runtime environment to interact with another one hosted at a remote machine over the network to coordinate and perform a particular task.

An Akka router is an actor which route messages to other actors called routees. The router actor is responsible for distributing tasks among its routees using different routing strategies. In this application, we used the round-robin-group routing algorithm in which messages are sent to routees in a round-robin fashion. The router actor can also deploy its created children on a set of remote hosts.

There are two types of routers: (i) Pools which are routers that can be provided a set of already created routees for its use. The programmer can pass the paths of the routees to the router’s configuration. Routers are supervised by its parent actors, and thus they are the supervisor of their children.

We used two laptops and one Raspberry Pi 3 device to test the developed application. The Raspberry Pi served as the master node and the two laptops served as worker nodes. First, the local runtime --which is deployed at the Raspberry Pi-- created a master actor, which instructs the router to create and deploy 10 child actors on the other two remote hosts in a round-robin fashion. In order to deploy routees remotely, we wrapped the router configuration in a RemoteRouterConfig by attaching the remote addresses and ports of the destination nodes.

Once the routees are deployed at the remote runtimes, they start executing the word-counting task and send results back to the master actor through the router. Particularly, worker actors read the input text line-by-line, count and index words before sending results back to the master actor. Once a new message arrives at the master node, the master actor forwards the indexed words to a mapper-aggregator worker which is responsible for mapping and aggregating these words from the two workers to display an up-to-date view of results, as shown in Figure 4.

V. Evaluation

In this section, we present our experimental evaluation of the runtime environment for performance and scalability. Our experiments were conducted on three heterogenous machines including Raspberry Pi and laptop devices. In these experiments, we used

---

3Raspberry Pi 3 is a single-board computing device with wireless LAN connectivity and a 1.4GHz 64-bit quad-core processor.
two computationally-intensive applications which can be smoothly parallelized to demonstrate certain performance characteristics of the actor model such as the rate of message processing and the computational time. We used one Raspberry Pi 3 device with a 1.2GHz quad-core ARM Cortex-A53 processor and 1GB of RAM running Raspbian OS. We also used two Windows laptops: the first is equipped with a 2.6GHz Intel Core™ i5-2540M CPU (4 cores) processor and 16GB of RAM, while the second is equipped with a 2GHz Intel® Core™ i3-5005U CPU (2 cores) processor and 8GB of RAM.

An actor can be in one of two states: idle and busy. Each actor hosted in our runtime environment remains in the idle state until it receives a new message which changes its behavior to the busy state. Once the message is processed, the actor’s behavior is changed back to the idle state. Therefore, in our experiments, we measured the time taken per message processed, beyond what was required for processing the message (i.e., carrying out the actual computation).

We used Scala version 2.13.0-M1 with Akka version 2.5.19 running on JVM 1.8. We set the minimum and the maximum number of active threads in the pool of threads, called parallelism-min and parallelism-max, to 8 and 64, respectively. The parallelism-factor is set to 8. The parallelism-factor is used to determine the thread pool size (i.e., the core number of threads) at start-up, using the formula:

\[
\text{ceil} = \text{available processor cores} \times \text{parallelism factor}
\]

The resulting size is then bounded by the parallelism-min and parallelism-max values. However, if a new task is submitted to the pool and there are fewer threads than the maximum pool size, an additional thread will be created as long as the maximum pool size is not exceeded. The parallelism-min, parallelism-max and parallelism-factor settings for each pool of threads provide a way to dynamically size these pools based on the number of CPU cores available. Also, each experiment was carried out 10 times.

Next, we describe the two experiments which we carried out to evaluate our system.

A. Monte Carlo Pi (\(\pi\)) Simulation

We carried out a simulation to estimate the value of \(\pi\) (3.141592...) by using the Monte Carlo method.\(^4\) We generated a large number of random points, then we checked how many of these arbitrary numbers fall in a circle enclosed by the unit square. The ratio of the number of points falling within the circle to the number of a point falling within the square multiplied by 4 is proportional to the value of \(\pi\). In other words, if we divide the number of points within the circle (\(N_{\text{circle}}\)) by the total number of points within the square (\(N_{\text{square}}\)), we get an approximate value of \(\pi/4\), as follows:

\[
\frac{\pi}{4} \approx \frac{N_{\text{circle}}}{N_{\text{square}}}
\]

where \(N\) is the number of generated points.

The number of points \(N\) used in the simulation is proportional to the accuracy of the result. When we use a small number of points, the estimation will not be very accurate. Therefore, we experimented with 100 million points, and gradually increased the number of actors contributing to this task. We distributed these intensive computations across the Raspberry Pi and the two laptops equally so that they collectively calculate an approximation of \(\pi\). As a result, we achieved an excellent approximation to the actual value – to within around 12 decimal places of accuracy.

We ran a set of experiments to determine the impact of changing the number of actors on computational CPU time. In the first experiment, we used one actor to perform all computations. Then, we gradually doubled the number of actors in the following experiments. Figures

\(^4\)Monte Carlo method is a statistical method that utilizes a sequence of random numbers to perform a simulation.
5 and 6 show the results at both the Core-i3 and Core-i5 laptops. Although the CPU time stays roughly between 1 and 2.6 seconds for the core-i5 laptop, it ranges from about 1.4 and 3 seconds for the core-i3 laptop.

As shown in the figures, as the number of actorsused in the computations increases, the total CPU time decreases until reaching 12 actors. At this point, we could not observe obvious differences in the computational time. This may be justified by the extra overheads of initiating the actors and communication delay. These results suggest that having a large number of actors is not necessary to improve the overall performance of intensive computations. Therefore, programmers need to find an equilibrium between the number of actors and the amount of CPU cycles required to carry out the calculations.

Figure 7 shows the impact of increasing the number of actors on computational CPU time at the Raspberry Pi 3. Similar to figures 5 and 6, this Figure 7 shows a decreasing trend between the numbers of actors and the total computational time. However, the figure shows that the curve diverges significantly between 2 and 12 actors. Figure shows this on a linear scale between these two points, where the divergence begins to happen around 2 actors, meaning that dividing the number of computations among more than one actor can achieve a better performance, especially for limited-resource devices such as the Raspberry Pi 3.

**B. Fibonacci Numbers Sequence Calculator**

We carried out another simulation to calculate the Fibonacci sequence which is the series of numbers (0, 1, 1, 2, 3, 5, 8, ...) following this simple mathematics rule:

\[ x_n = x_{n-1} + x_{n-2} \]

where \( n \) is the \( N \)th Fibonacci position and the \( N \)th Fibonacci position is found by adding up the two numbers before it.

As explained in the previous section, routers create new routees and send messages to them in a round-robin fashion. Figure 9 shows a code snippet illustrating how a RoundRobinRouter would be created and allocated 100 FibonacciActor routees for its use to perform the calculations. The figure also shows the implementation of the FibonacciActor as a simple tail recursive Scala function.

We ran a set of experiments to determine the effect of changing the \( N \)th Fibonacci position on the total computational time. In this experiment, we calculated the Fibonacci numbers corresponding to \( N \)th Fibonacci position for large values of \( N \) of up to 1M. We distributed these intensive computations across the Raspberry Pi and the two laptops equally so that they collectively calculated the target position. Specifically, we used the Raspberry Pi device as the master node and the two
laptops as worker nodes. The computation started by sending a message to the master node where the master actor divided the computation into three parts. Two parts are sent to the worker nodes, and one part was executed locally. The master actor was responsible for mapping and aggregating the intermediate results from the two workers, and then displaying the final result.

Figure 10 shows the results at both the Raspberry Pi 3 and the two laptops. The figure shows an increasing trend between the Nth Fibonacci position and the CPU computational time. Although the CPU time stays roughly under 15 seconds for the core-i3 laptop and stays roughly under 13 seconds for the core-i5 laptop, it jumps roughly under 15 seconds for the core-i3 laptop and stays CPU computational time. Although the CPU time stays ing trend between the Nth Fibonacci position and the 3 and the two laptops. The figure shows an increas-
and aggregating the intermediate results from the two machines. This experiment demonstrates the benefits of leveraging the high-performance computing power of worker machines to process this highly-intensive

VI. Conclusions

In this paper, we presented an approach to support the efficient execution of actor-based applications in Akka. Particularly, we described our design and implementation of a distributed runtime environment over which such class of distributed applications could be implemented relatively easily across heterogeneous computing machines. We evaluated the programmability benefits of our system by implementing a number of distributed applications running on heterogeneous devices. We also carried out several sets of experiments for evaluating the performance and scalability of our system, paying particular attention to establishing the relationship between the distribution of computations and the total computational time for executing them. The results showed that the computational time depends on various granularity characteristics of the systems, most notably the sizes of the computations assigned to individual machines.

REFERENCES