An Actor-Based Distribution Model for Realtime Interactive Systems

Stephan Rehfeld*
Beuth University of Applied Sciences

Henrik Tramberend†
Beuth University of Applied Sciences

Marc Erich Latoschik‡
University of Würzburg

Abstract

This article illustrates the design and development of distribution properties for Realtime Interactive Systems (RIS). The approach is based on the actor model for concurrent computation. The actor model provides a unified API for intra-node as well as for inter-node distribution and strongly facilitates the development of concurrent applications. Several benchmarks analyze vital performance properties to support the design decisions taken. The benchmarks describe typical setups found in RIS-applications, i.e., distributed rendering for large screen and tiled displays in immersive VR setups. Actual and potential performance impacts caused by the actor middleware are analyzed and identified and alternative solutions to overcome these impacts are provided.

Index Terms: I.3.2 [Computer Graphics]: Graphics System—Distributed/network graphics

1 Introduction

Typical application areas of Realtime Interactive Systems (RIS) span from Virtual Reality (VR), Augmented Reality (AR), multimodal human-computer interaction (HCI), robotics, to computer games. A main characteristic of such systems is the integration of multiple functional properties to analyze live user input and to synthesize appropriate and consistent output in real-time. A common task during RIS developments is the integration of existing software modules and libraries that provide the required functional properties. They most likely will use incompatible data structures and control flows.

RIS-related integration tasks increasingly face distribution requirements. The computational power of modern computer hardware is commonly achieved by so-called *intra-node* parallelism of multicore CPU and GPU architectures. Additionally, *inter-node* distribution based on compute clusters plays an important role for high performance computing and the RIS area. For example, a common requirement of various VR systems is the distributed rendering of a coherent virtual scene. But developing distributed software has its own pitfalls like dead locks, live locks, heisenbugs, or stochastic behavior which complicate development drastically.

This article describes and analyses the actor model [13, 2] as an alternative RIS distribution model. It is applicable for intra-node as well as inter-node distribution and addresses many of the typical problems associated with distributed computing. The article starts with the reflection of related work followed by a brief description of the software platform [16, 15] which is utilized to implement a unified distribution API based on the actor model. Several benchmarks provide comprehensive results which compare the inter-node efficiency to the intra-node efficiency. The benchmarks expose an unexpected and severe bottleneck. This bottleneck is identified by subsequent low-level benchmarks leading to a solution presented at the end.

*e-mail: rehfeld@beuth-hochschule.de
†e-mail: tramberend@beuth-hochschule.de
‡e-mail: marc.latoschik@uni-wuerzburg.de

2 Related Work

Intra-node distribution, aka *clustering*, has been a well-proven concept to provide the computational power for RIS applications [25, 6, 4, 18, 22]. Typically, clustering is used to serve multiple stereoscopic displays or to perform highly complex computational simulations in real time.

Historically, dedicated computer systems like the SGI ONYX series have been used in this domain. They utilized a parallel CPU and graphics hardware design with inter- and intra-node characteristics and provided a dedicated API to handle the expensive hardware. Since the development of high-quality consumer graphics in the PC market, inter-node clustering of Of-The-Shelf (OTS) equipment became a cost-efficient alternative to dedicated computer systems.

In addition, increasing computational power by raising CPU clock frequencies became uneconomical [23, 24] due to heat production. Since the beginning of the 2000's computational power is mainly increased by parallel on-die architectures utilizing multiple CPU cores. These multicore CPUs are the dominant architecture for PC hardware as of today (2013) [20, 3].

The advent of multicore CPUs has a large impact on mainstream software development down to the consumer level. As a result, distribution became increasingly important for the gaming area as well. Sophisticated games have to take advantage of intra-node as well as inter-node distribution. The latter is important for online gaming and massive multiplayer games [12, 22]. The clustering paradigms usually applied in such scenarios are n:1 or N:m client-server models where central game servers collect and distribute the changing game states to and from clients.

Developing reliable and maintainable multithreaded software is a highly complex task. Dead locks, live locks, heisenbugs, and stochastic behavior are just a few typical errors types frequently encountered during the development of multithreaded software. The classical *shared state concurrency* where multiple threads working on the memory and synchronizing with each other by using locks, semaphores, or signals turned out be unmanageable for large software [17]. Just having multiple cores to one's availability or to interconnect a bunch of PCs does not provide a means to take advantage of the computational power. Software models are required which have to encompass inter- as well as intra-node concurrency and clustering setups.

Hewitt's actor model [13, 2] provides a promising programming paradigm to write reliable and maintainable concurrent software. Actors are independent flows of control. Each actor has its own protected memory area, which can not be read or written by other actors. Actors solely communicate with each other via messages. The behavior of an actor describes an actor's reaction to a message. An actor can manipulate its own state, change its own behavior, send messages to other actors, and spawn more actors.

Message passing is asynchronous. If an actor sends a message, the sender does not have to wait until the receiver processes it. If the receiver currently is busy, then the sent message is stored in the mailbox of the receiver. The receiver retrieves the next message from the mailbox after it finishes its current task.

A broad range of problems with different granularity can be encapsulated into actors. This makes the actor model highly adaptable to many application scenarios. The main advantage of the actor model is that it abstracts concurrency into a usable programming

model and avoids common pitfalls of concurrent programming. It provides a unified approach that covers both, intra-node as well as inter-node scenarios. For example, in the RIS context it can greatly simplify transitions between clustered and local versions while always utilizing the underlying hardware economically.

Beside several advantages, the actor model has to be analyzed in terms of potential disadvantages. Messages need to be created, saved in the mailbox of the receiver, and decoded by the receiver before processing it. Clearly, this is more expensive than a simple function call. At first glance this causes an unnecessary overhead for intra-node distribution. For inter-node distribution, the underlying transport layer may have additional performance impacts. Also, the asynchronous behavior of the underlying communication model must be augmented by a synchronization layer to support coherent computation for several RIS-specific tasks, e.g., synchronized rendering across a cluster.

The remainder of this article will analyze the actor model in terms of its functional and performance characteristics with respect to RIS applications.

3 Platform Layer

The actor-based distribution model has been integrated into the platform API of Simulator X [16, 15], an experimental RIS middleware implemented in Scala. Scala is a modern multi-paradigm language that contains object-oriented and functional language features [19]. Scala uses the actor model as the fundamental concurrency model and provides its own native actor implementation [11]. Several alternative actor model implementations [1] exist. As light-weight concurrent constructs, multiple actors are dispatched to heavy-weight threads of the underlying operating system by an internal actor scheduling system. Typically, one heavy-weight thread exists per CPU core, while several thousand actors may be created.

The Scala actor model library provides fundamental inter-node abilities. Actors within the local process can be registered to the *remote actor service*. Another process can connect through the network and can create a local proxy of a registered actor. The proxies are called *remote actors* and can be used like any local actor. If a message is sent to a remote actor, it is serialized and sent over the network.

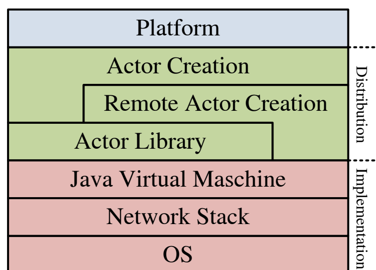

Figure 1: The overall architecture of the inter-node subsystem.

Figure 1 illustrates the overall architecture of the RIS middleware and the location of the actor-specific concepts. The central programming concepts of the top-level platform API—components, entities, and state-variables—and their integration with the actor model are briefly described here.

Components

Components provide essential services for RIS applications within Simulator X, e.g., rendering, physics simulation, or artificial intelligence, etc. Components are functional high-level building blocks providing different aspects of a simulation. During entity construction, components interpret associated entity descriptions and they create state variables and initial values for properties relevant to a components functionality. Components are implemented as actors providing coarse-grained concurrency.

Entities

Entities encapsulate visible as well as invisible simulation objects. They aggregate associated state variables. An entity can have state variables for attributes like transformation, mass, surface friction, and shininess. Each represents an attribute of the simulated object that can be managed by different actors in the system. State variables like mass or surface friction are managed by an actor that performs the physics simulation. Rendering attributes like shininess are managed by an actor that renders the scene. For example, a physics engine and a graphics engine typically communicate via a *transformation* state variable. The physics engine writes a new transformation to the variable and the graphics engine updates its scene representation. The set of the entities of all simulated objects provides a pseudo-global world state.

State variables

State variables provide the changeable state of a simulation. The value change operation of state variables is built on top of the underlying message-passing system of the actor model. State variables are typed handler constructs to read, write, and observe variables that are shared between actors. They represent an attribute of a simulated object or a parameter of a system device like a key on the keyboard. State variables are the finest-grained concurrency concepts integrated into the platform API.

- Reading a state variable means requesting the current value of the variable from the managing actor.
- If an actor wants to write to a state variable, it sends a message that contains the new value to the managing actor.
- Observing a state variable means that the managing actor notifies the observer on value changes.
- Writing to a state variable is interpreted by the managing and observing actors.

4 Requirements of the Distribution Layer

Components and state-variables provide appropriate targets for the integration of the required distribution functionality into the platform API. Several requirements have been identified in the relevant literature on VR systems [5, 25, 21] and from analyzing the needs of application programmers. Ideally, the desired distribution functionality for intra- and inter-node distribution should fulfill all of the following requirements:

- **R1**: *Distributability*—Non-distributed applications must be easy to be migrated to clustered set-ups.
- **R2**: *API-uniformity*—The distribution API must be similar for intra-node as well as inter-node applications.
- **R3**: *Compatibility*—An application that works locally on one node should also work distributed in a cluster.
- **R4**: *Flexibility*—Beside actors that have access to specific hardware, the developer should freely decide which actor runs on which node of the cluster.
- **R5**: *Configurability*—Simple configuration of the clustering subsystem and the distribution layout.

- **R6**: *Transparency*—As development of an application is mainly done using a normal desktop computer, a development mode that disables the cluster feature needs to be provided.
- **R7**: *Manageability*—Low administration overhead.

5 DISTRIBUTION LAYER

This section describes the architecture of the distribution layer. A cluster for inter-node distribution is defined here as a networked set of computing nodes. Each of the nodes has a unique name. At start-up, the nodes connect to each other. The application starts after all nodes are connected. Figure 1 illustrates the parts and layers of the distribution model and how they are integrated into an existing platform. From bottom up: The first three layers—OS, network stack, and actor library—are provided by the operating system and the Scala virtual machine. Creating an actor on a foreign node is implemented by the *remote actor creation* layer. To create an actor, it uses the actor library.

All actors in the cluster can create actors on any other node. All nodes have the same rights, and *no master* node exists. Every node in the cluster is connected to every other node in the cluster (Figure 2) by permanent TCP connections.

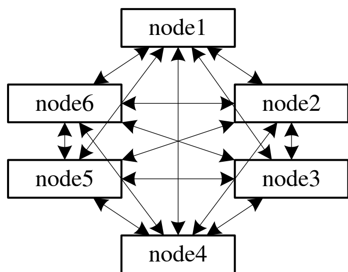

Figure 2: All nodes in the cluster are connected to each other.

Actor creation

Creating actors on other nodes is a key feature of the inter-node subsystem, but not originally supported by the actor library. To create actors, a function called *createActor* is provided. Two overloaded versions exist, one to create an actor on the local system, and one to create an actor on a foreign node. Listing 1 shows how two actors of the same type are created on the local and a foreign node.

Two actors are created in lines 1 and 3. Line 1 creates an actor on the local node. The values of a and b are passed as parameters for the constructor. Line 3 creates an actor of the same type on a foreign node of the cluster.

Conforming to requirement **R2** (*API-uniformity*), both calls only differ by one parameter, the optional name of the target node. If no name is passed, the actor is created on the local node. If a name is passed, the actor is created on the specified node in the cluster. Conforming to requirement **R5** (*Configurability*), the distribution layout of an application can be changed by simply adding a name of a node or changing the name of the current target node. This is the only point where local and remote actors are treated differently. After the actor is created, the API to interact with both actors is identical. Passing an optional name for a target node is the absolute minimum difference. Hence, a non-distributed application is easy to be migrated (**R1** *Distributability*). According to requirement **R3** (*Compatibility*), the application developer is free to choose if he wants to create an actor on another node as he could also create both actors on the local node or each one individually on any other node in the cluster. Conforming to requirement **R4** (*Flexibility*), the actors that are created on a foreign node do not need to extend any special base class but the base class of the underlying actor model library.

Listing 1: Creating two actors using the distribution API. The first actor is created on the local node. The second actor is created on another node in the cluster.

```
1  createActor[ActorType]( a, b )
2    ( handler )()
3  createActor[ActorType]( a, b )
4    ( handler, 'nodeB )()
```

Abstract class for applications

The distribution layer provides the abstract class `SimXClusterApplication` to facilitate the development of intra-node applications. According to requirement **R7** (*Manageability*), it interprets a simple set of command line arguments to configure the distribution layer. In accordance to requirement **R6** (*Transparency*), the inter-node distribution can be disabled by one parameter. A single function call is necessary in the API to configure and boot the cluster. To conclude, an inter-node capable application differs from an intra-node application by the application base class used and one additional line in the source code. As the inter-node subsystem can easily be disabled, inter-node applications can run as a intra-node application without additional work.

Programming

Listing 2 shows an example of an inter-node application. The applications class needs to extend the `SimXClusterApplication` class. Lines 1 and 2 show that the base class consumes an `Array` of `String` as an argument. This should be the command line arguments array handed over to the *main function* by the JVM. Lines 4 and 5 configure and boot the cluster. The setter method *clusterConfiguration* takes a set that contains the names of all nodes in the cluster as parameter. This method blocks until all nodes have joined the cluster. Because this function is called within the class definition body, the creation of an instance via the `new` operator in the main function blocks until the cluster is started.

The main function is shown in lines 10 – 13. A new instance of the `BootExample` class is created and the command line arguments are passed as parameters. The line *bootExample.start()* starts the actor. At this point the cluster is already set up and all nodes are connected to each other.

Listing 2: Small example that uses the abstract class `SimXClusterApplication`

```
1  class BootExample( args : Array[String] )
2    extends SimXClusterApplication( args ) {
3
4    clusterConfiguration = Set() + 'simulation
5      + 'left + 'right
6    ...
7  }
8
9  object BootExample {
10   def main( args : Array[String] ) {
11     val bootExample = new BootExample( args )
12     bootExample.start()
13   }
14 }
```

Example: Distributed graphics rendering

A typical distribution scenario for immersive VR systems requires multiple rendering nodes to operate on commodity hardware. Listing 3 illustrates a display setup description for a powerwall with two video projectors connected to two different nodes. The nodes have already been initialized following Listing 2.

This display setup description is interpreted by the rendering component that creates actors on every cluster node connected to a described surface. The actors on the node open required windows on the specified ports to render the scene.

Listing 3: A display description that opens two windows on different nodes in a cluster.

```
1  val pos = ConstMat4f(
2      Mat3x4f.translate(
3        Vec3f( 0.0f, 0.0f, -1.0f )
4      )
5    )
6  val displayDesc1 = new DisplayDesc(
7    resolution = None,
8    size = (3.73, 2.33),
9    transformation = pos,
10   view = new CamDesc( 0, Eye.LeftEye )
11 )
12 val displayDesc2 = new DisplayDesc(
13   resolution = None,
14   size = (3.73, 2.33),
15   transformation = pos,
16   view = new CamDesc( 0, Eye.RightEye )
17 )
18 val displayDevice1 = new DisplayDevice(
19   hardwareHandle = None,
20   displayDescs = displayDesc1 :: Nil,
21   linkType = LinkType.SingleDisplay,
22         // Node for the left eye
23   node = Some( 'left )
24 )
25 val displayDevice2 = new DisplayDevice(
26   hardwareHandle = None,
27   displayDescs = displayDesc2 :: Nil,
28   linkType = LinkType.SingleDisplay,
29         // Node for the right eye
30   node = Some( 'right )
31 )
32 val displaySetupDesc = new DisplaySetupDesc()
33 displaySetupDesc.addDevice(
34   displayDevice1, 0 )
35 displaySetupDesc.addDevice(
36   displayDevice2, 1 )
37 renderer ! ConfigureRenderer(
38   displaySetupDesc,
39   ...
40 )
```

6 Distributed Rendering Benchmarks

This section describes the benchmarks and results to evaluate the distribution model. The benchmarks were first performed using intra-node distribution to obtain base-line values for the later performed inter-node versions.

Scenario

In the benchmark application illustrated in Figure 3 the model of a medieval city is rendered. The user can spawn single barrels and a stack of barrels in the middle of the scene. A large barrel can be pushed forward to collide with the stack of barrels.

The application consists of two components. The physics component simulates the movement and collision of the barrels. The renderer renders the scene. The physics component performs the whole simulation within one actor. The rendering component manages several *render actor*s. A render actor opens one window and renders the scene to this window. One render actor runs on every node that is connected to a display or video projector. Typically, each of the render actors render a different view frustum, e.g. for the left the right eye or for a part of a tiled display. The render actors observe the transformation state variables of the entities of the scene. Hence, the communication between the physics component and the render actors are the updated transformations of the entities. Ten barrels are simulated in the performed benchmarks. Including trigger messages, 720 messages per second are sent to each render actor.

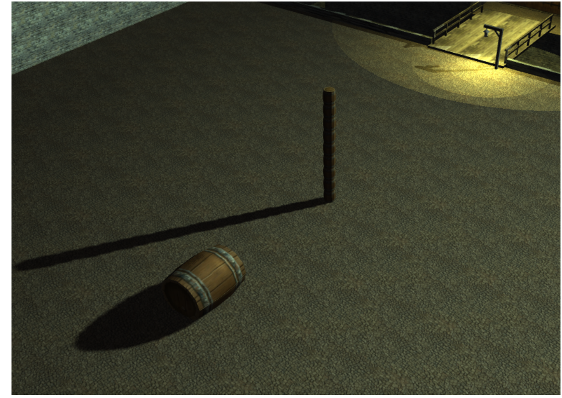

Figure 3: Screenshot of the barrel stack benchmark application.

Setups

Two typical setups are covered by the benchmarks:

- A high quality powerwall, where the image for each eye is rendered by one node.
- A 3x3 tiled display, where each tile of the assembly is connected to one node.

The powerwall setup measures the latency and synchronization between the nodes. A high latency or failed synchronization between the nodes—where both nodes render different world states—will result in a failure of the stereoscopic impression.

The rendering and simulation hardware of the powerwall consists of the three computers (Table 1). The video signals are fed to two projectors that project the images onto the same screen using a KVM switch and DVI splitters. All three computers are connected by a fully switched 1 GBit Ethernet. The tiled display setup measures the scalability of our distribution model. Ten computers were used for the tiled display tests (Table 2). All ten computers are connected by a fully switched 1 GBit Ethernet.

Table 1: Hardware of the powerwall.

	simulation server	render node
CPU	2x AMD Opteron 6168 12-Core 1.9GHz	AMD FX-6100 6-Core 3.3GHz
RAM	16 GB	8 GB
graphics card	2x AMD Radeon HD 6950 1GB	AMD Radeon HD 6950 1GB

Table 2: Hardware of the tiled display.

CPU	Intel Core 2 CPU 6600 2.4 GHz
RAM	6 GB
graphics card	NVIDIA GeForce 9600 GT

VSYNC

The benchmark application is designed to run with at least 60 fps on the used hardware. Still, the frame rate will never exceed 60 fps due to activated VSYNC. Higher frame rates could be achieved without VSYNC but hardware and view frustum dependent effects would interfere with the target measurements as a result. In the tiled display scenario each tile renders a different view frustum. Each view frustum contains frustum-specific geometry of individual complexity and materials. Hence, tile-specific frame rates due to individual rendering processes would interfere with measurements of the inter-node distribution. This problem would even be potentially exacerbated due to the varying hardware of both scenarios, which would lead to results hardly comparable to each other in case of disabled VSYNC.

Configuration

Under both setups the benchmarks were performed with different configurations. In both setups an intra-node version were executed to generate a comparison value. In the intra-node version the physics component, the rendering component, and one render actor runs on the same node. This comparison value shows how many fps are achieved without inter-node distribution.

In the inter-node powerwall setup one node performs the simulation (e.g. physics simulation) while up to two nodes perform the rendering. The physics and the render component run on the simulation server, while the two render actors each run on one render node. In the inter-node tiled display setup one node performs the simulation and 9 nodes perform the rendering. Again, the physics and the render components run on the simulation node, while each of the render actors run on individual nodes of the cluster.

Additionally, the synchronization between the nodes is modified. In the synchronized configuration all render actors on the nodes are synchronized to the rendering component actor on the simulation node using the standard actor message passing. The rendering component on the simulation node triggers the rendering nodes and waits until they complete the rendering. Then the next frame is rendered. In the unsynchronized configuration every render actor triggers itself after it finishes a frame. The rendering actors just render the last known state and trigger themselves after processing all messages that are stored in the queue while rendering.

Results

The results of the distributed rendering benchmarks are summarized in tables 3 and 4.

Table 3: Results of the powerwall benchmark scenario.

	intra-node	inter-node (2 Render nodes)
sync	60	5
no sync	60	60

Table 4: Results of the tiled display benchmark scenario.

	intra-node	inter-node (9 Render nodes)
sync	60	5
no sync	60	60

Powerwall intra-node sync/unsync

Over the full runtime of the application, the average frame rate was 60 fps. This is the expected frame rate and confirms the results of existing demo applications[8, 9, 10, 7].

Powerwall inter-node sync

This benchmarks achieved a frame rate of 5 fps. This low frame rate was not expected after the high frame rate of the intra-node benchmarks.

Powerwall inter-node unsync

With disabled synchronization between the render actors and the renderer component actor the frame rate rises to 60 fps on all nodes.

Tiled display intra-node sync/unsync

Like in the *Powerwall intra-node sync/unsync* benchmarks, the average frame rate was 60 fps. Also, this frame rate is the expected value.

Tiled display inter-node sync

Like in the *Powerwall inter-node sync* benchmark, the average frame rate was 5 fps. As *Powerwall inter-node sync* and *Tiled display inter-node sync* were executed on different hardware, the low frame rate in a inter-node scenario does not depend on the used hardware.

Tiled display inter-node unsync

Like in the *Powerwall inter-node sync*, the frame rate of all nodes rise to 60 fps without synchronization. The synchronization between the render actors and the renderer component seems to be the source of the low frame rate.

Overall discussion

While a high frame rate is achieved for intra-node setups with disabled synchronization, enabled synchronization leads to a low frame rate for inter-node setups. Disabling the synchronization in general is not acceptable, as the nodes often render different world states. This is most notable in *Powerwall inter-node unsync*, as the images for both eyes are not in sync and the stereo impression fails for moving objects.

To identify the causes for the poor performance given by the previous cases, we formulate three hypothesis about which layer causes the low frame rate. Each of them covers one of the layers illustrated in Figure 1.

- **H1** A performance problem of the platform layer.
- **H2** A performance problem of the distribution layer.
- **H3** A performance problem of the implementation layer.

7 Low-Level benchmarks

The following low-level benchmarks test the throughput and latency of the system under different conditions and aspects. The benchmarks were executed using different configurations in order to try to falsify the three hypothesis. To falsify **H1** the benchmarks were implemented on top of the platform layer and on top of the underlying distribution model implementation–the Scala actor model library. To falsify **H2** the benchmarks were reimplemented upon Akka, an alternative distribution model implementation. As the causes of the poor performance could already be located in these

configurations, there was no need to falsify **H3**. However, if necessary, it would require a reimplementation of the benchmarks using a different distribution model.

Scenarios

Throughput

The throughput benchmark scenario measures the maximum throughput of messages per second over the network. As illustrated in Figure 4, two types of actors are involved. The *sender* sends messages to a *receiver*.

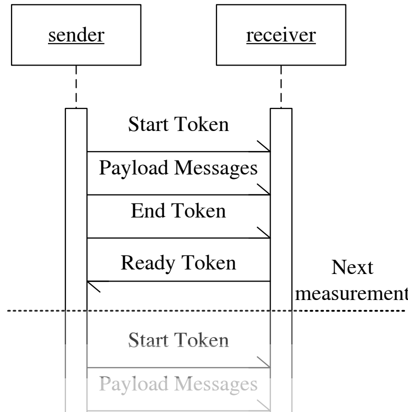

Figure 4: Communication order in the throughput benchmark scenario.

The *sender* sends a start token to the *receiver* at the beginning of each measurement . The start token contains the number of messages that will be sent in this step. Afterward, the *sender* sends the messages with a given payload. After sending the payload messages, it sends an end token. On processing the start token, the *receiver* saves its own local start time. The payload messages are just read from the inbox queue. On processing the end token, the *receiver* calculates the time span and logs it.

Next, the *receiver* sends a ready token to the *sender*. After the *sender* receives the ready token, it starts the next step.

Ping

The ping benchmark tests the latency between two actors that are communicating in an inter-node configuration. One actor receives a *ping* message and sends back a *reply* message to the sender (Figure 5). After the sender received the *reply* message it immediately sends the next ping message. The latency is determined by calculating the time span between sending the ping and receiving the reply. The benchmark ran for 2 minutes and the average was calculated for this time span.

Setup

All benchmarks were executed on the powerwall hardware. In the inter-node throughput scenario, the sender actor runs on the simulator server, while the receiver runs on the render node hardware. In the ping scenario, the sender runs on the simulation server, while the reply actor runs on the render node hardware.

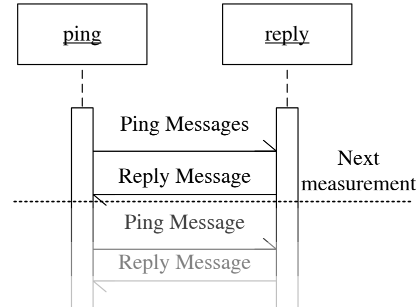

Figure 5: Communication order in the ping benchmark scenario.

Configuration

Both, the ping and the throughput benchmarks were implemented once on top of the platform layer and circumventing the platform layer. The latter were directly realized on top of the native Scala actor model library and on top of Akka, an alternative actor model implementation. The throughput benchmarks were executed locally on the simulation serverand between the simulation server and one render node.

A payload of 64 bytes was used, which represents a 4x4 transformation matrix consisting of single precision float values.This payload was choosen, because transferring transformation matrices is one of the most common tasks within an interactive 3D application. For example, it is used to update the position of simulated objects, of input devices like a WiiMote with a tracking marker, or of the user's head position.

Results

The results of the low-level benchmarks are summarized in tables 5 and 6. The *Throughput intra-node Scala* benchmark was not performed, because the performance in intra-node setups is good and the focus were put on inter-node setups.

Table 5: Results of the throughput benchmarks in messages per second.

	Platform	Scala	Akka
intra-node	120,000	—	1,140,000
inter-node	11,400	11,400	90,000

Table 6: Results of the ping benchmarks in seconds.

Platform	Scala	Akka
0.2	0.2	< 0.001

Throughput intra-node platform layer

The throughput of the platform layer is about 120,000 messages per second, locally on one node.

Throughput intra-node Akka

The throughput of Akka is about 1,140,000 messages per second, locally on one node. This is about 10 times more than in the Scala actor model library.

Throughput inter-node platform layer

The throughput of the platform layer in an inter-node setup with one node is 11,400 messages per second. Sending messages over the network to a remote actor is orders of magnitude slower than on the local node. The measured throughput is lower than expected. But it does not cause the low frame rate in the distributed rendering benchmarks, because only 720 messages per seconds are sent to each render actor.

Throughput inter-node Scala

The throughput benchmark on top of the Scala actor model library yields the same result as *Throughput inter-node platform layer*. This benchmark shows that the low throughput is not caused by platform layer, but by its underlying actor model library. So, the hypothesis **H1** is wrong.

Throughput inter-node Akka

The throughput benchmark on top of Akka reaches 90,000 messages per second. The throughput is about 10 times higher than in the platform layer and the Scala actor model library. This benchmark shows, that the hypothesis **H2** is correct, as the low throughput is caused by the Scala actor model library.

Ping platform layer

A ping and reply needs 0.2 seconds on average. An ICMP ping between the same node needs less than 1ms. This high latency in an inter-node setup causes the low frame rate, as only 5 pings can be send within one second. Sending messages to an actor on another node introduces a high latency.

Ping Scala

This benchmarks yields the same results as *Ping platform layer*. This benchmark shows that the high latency is not caused by platform layer, but by its underlying actor model library. So, the hypothesis **H1** is wrong.

Ping Akka

A ping on top of the Akka actor model library needs less than 1 ms on average. The latency is very close to the latency of an ICMP ping. This benchmark shows, that the hypothesis **H2** is correct, as the high latency is caused by the Scala actor model library.

Overall discussion

Communication between two actors over a network is certainly much more expensive than local communication on a node, as the benchmarks confirmed. While 120,000 messages per second can be sent within one node, only 11,400 messages per second are possible over the network in the platform layer and the Scala actor library. Akka allows about 10 times more messages per second. The throughput in the platform layer and the Scala actor model library is low, but not the reason of the low frame rate in the distributed rendering benchmarks.

It turned out, that the Scala actor model library introduces a high latency of 0.2 seconds, while Akka only has less than 1 ms latency. We identified the high latency as the reason for the low frame rate of the distributed rendering benchmarks. The low-level benchmarks show that the low throughput and high latency is not caused by the paradigms and API of the platform layer, but by the chosen underlying actor model library. The concurrency concepts of the platform layer, the components and state variables, do not add extra latency to the system.

The latency and throughput highly depends on the used concrete low level implementation. A latency close to the latency of the underlying network connection is usable for many application scenarios. The throughput of message might become a problem. The current demo applications work well after replacing Scala's native actor library by Akka.

8 Conclusion

We described and analyzed the actor model as an alternative RIS distribution model applicable for intra-node as well as inter-node distribution. Several benchmarks provided comprehensible results which compared the inter-node efficiency to the intra-node efficiency. The benchmarks identified an unexpected bottleneck. The roots of this bottleneck were identified by a series of low-level benchmarks and a solution was presented.

The latency measured is close to the latency of the underlying network technology. The latency that is introduced by the distribution models and the additional layers is minimal.

The actor model as a programming model is a good foundation for RIS. It abstracts concurrency in a very usable paradigm. The distribution API presented here provides a unified programming model for intra- and inter-node distribution. Both types only differ during bootstrapping and by one additional parameter for the actor creation function. The latter can even completely be eliminated by integrating an external configuration that contains the target node for each actor in the system.

Abstraction is highly beneficial and can make the development of reliable concurrent software manageable. However, abstraction does not release developers from the responsibility to perform tests and benchmarks on every abstraction layer carefully, to identify bottlenecks and unforeseen behavior of abstraction layers. Developing concurrent software becomes easier with the right abstractions, but utilizing concurrency for performance gains continues to be a demanding task.

9 Further research

Since latency is mission critical with respect to synchronization, there are two promising options to further improve performance of the distribution model. First, alternative transport layers, e.g., network technologies like InfiniBand, provide a much lower latency for inter-node setups out of the box. Akka already supports the exchange of the transport layer and hence is the right choice in that sense. Second, synchronization messages usually are very specific. They require very low latency but the usually do not need any payload. They are tailored signals. Tailored hardware connections, e.g., based on USB or LPT ports, with built-in hardware priority could be used. They would replace the heavy-weight message de/serialization and dispatching by only one electric signal while they could be hidden behind the uniform API of the message system which is easily achieved by Scala's extension mechanisms.

In addition, the message transmission could be optimized by an optimized serialization and message dispatching. In all benchmarks, the standard de/serialization mechanism of the underlying JVM was used. An optimized dispatching mechanism may improve the performance even more. Scenarios with an extensive demand for a high number of messages will require additional message caching and aggregation facilities cumulating multiple small messages of lower payload into fewer fat messages of higher payload.

During the performed low-level benchmarks we did not performed any benchmarks to falsify **H3**, as we already identified **H2** as the reason for the poor performance. However, the results of benchmarks that examine the performance in contrast to implementations of alternative distribution models would give valuable information to evaluate the general performance of the actor model in contrast to these alternatives.

ACKNOWLEDGEMENTS

The authors wish to thank Dennis Wiebusch, Martin Fischbach, Hartmut Schirmacher, and Pedram Merrikhi. This research was supported by the German Federal Ministry of Education and Research program *Ingenieur*Nachwuchs, project SIRIS (#17N4409)

REFERENCES

[1] Comparison between 4 actor frameworks `http://doc.akka.io/docs/misc/Comparison_between_4_actor_frameworks.pdf`.

[2] G. Agha. *Actors: A Model Of Concurrent Computation In Distributed Systems*. PhD thesis, MIT Artificial Intelligence Laboratory, 1985.

[3] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54:67–77, May 2011.

[4] M. Bues, T. Gleue, and R. Blach. Lightning: Dataflow in motion. In Latoschik et al. [14], pages 7–11.

[5] L. Deligiannidis. *Dlove: a specification paradigm for designing distributed vr applications for single or multiple users*. PhD thesis, Tufts University, Medford, MA, USA, 2000. AAI9955979.

[6] L. Deligiannidis and R. J. Jacob. Improving performance of virtual reality applications through parallel processing. *J. Supercomput.*, 33:155–173, September 2005.

[7] M. Fischbach, M. Latoschik, G. Bruder, and F. Steinicke. smartbox: Out-of-the-box technologies for interactive art and exhibition. In *Virtual Reality International Conference VRIC*. ACM, 2012.

[8] M. Fischbach, D. Wiebusch, A. Giebler-Schubert, M. Latoschik, S. Rehfeld, and H. Tramberend. Sixton's curse- simulator x demonstration. In *Virtual Reality Conference (VR), 2011 IEEE*, pages 255 –256, march 2011.

[9] M. Fischbach, D. Wiebusch, M. E. Latoschik, G. Bruder, and F. Steinicke. Blending real and virtual worlds using self-reflection and fiducials. In M. M. Marc Herrlich, Rainer Malaka, editor, *ICEC*, volume 7522 of *Lecture Notes in Computer Science*, pages 465–468. Springer, 2012.

[10] M. Fischbach, D. Wiebusch, M. E. Latoschik, G. Bruder, and F. Steinicke. smartbox a portable setup for intelligent interactive applications. In O. D. Harald Reiterer, editor, *Mensch & Computer Workshopband*, pages 521–524. Oldenbourg Verlag, 2012.

[11] P. Haller and F. Sommers. *Actors in Scala: Concurrent programming for the multi-core era*. Artima, first edition edition, Jan 2012.

[12] M. Henning. Massively multiplayer middleware. *Queue*, 1:38–45, February 2004.

[13] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[14] M. E. Latoschik, D. Reiners, R. Blach, P. Figueroa, and R. Dachselt, editors. *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2008 workshop*. Shaker Verlag, 2008.

[15] M. E. Latoschik and H. Tramberend. A scala-based actor-entity architecture for intelligent interactive simulations. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2011 workshop*, 2011.

[16] M. E. Latoschik and H. Tramberend. Simulator X: A Scalable and Concurrent Software Platform for Intelligent Realtime Interactive Systems. In *Proceedings of the IEEE VR 2011*, 2011.

[17] E. A. Lee. The problem with threads. *Computer*, 39:33–42, May 2006.

[18] J.-D. Lesage and B. Raffin. High performance interactive computing with flowvr. In Latoschik et al. [14], pages 13–16.

[19] M. Odersky. The scala language specification version 2.9, may 2011.

[20] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3:26–29, September 2005.

[21] B. Raffin and L. Soares. Pc clusters for virtual reality. In *Proceedings of the IEEE conference on Virtual Reality*, VR '06, pages 215–222, Washington, DC, USA, 2006. IEEE Computer Society.

[22] A. Steed and M. F. Oliveira. *Networked Graphics: Building Networked Games and Virtual Environments*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[23] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

[24] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3:54–62, September 2005.

[25] H. Tramberend. Avocado: a distributed virtual reality framework. In *Virtual Reality, 1999. Proceedings., IEEE*, pages 14 –21, mar 1999.