Profiling and benchmarking event- and message-passing-based asynchronous Realtime Interactive Systems

Stephan Rehfeld [*]
Beuth Hochschule für Technik Berlin

Henrik Tramberend [†]
Beuth Hochschule für Technik Berlin

Marc Erich Latoschik [‡]
Universität Würzburg

Abstract

This article describes a set of metrics for a message-passing-based asynchronous Realtime Interactive System (RIS). Current trends in concurrent RISs are analyzed, several profiling tools are outlined, and common metrics are identified. A set of nine metrics is presented in a unified and formalized way. The implementation of a profiler that measures and calculates these metrics is illustrated. The implementation of an instrumentation and a visualization tool are described. A case study shows how this approach proved beneficial during the optimization of latency of an actual system.

CR Categories: C.4 [Performance of Systems]: Measurement techniques D.2.8 [Software Engineering]: Metrics—Performance measures I.3.2 [Computer Graphics]: Graphics Systems—Standalone systems

Keywords: profiling, benchmarking, concurrency, message passing

1 Introduction

Performance gains of modern computing environments largely depend on concurrency. Realtime Interactive Systems (RISs) in the areas of Virtual and Augmented Reality are exceptionally performance hungry. They have to provide highly responsive interaction cycles with low latencies. In addition, they combine multiple input and output channels, often with an advanced application logic and hence tend to exhibit complex software architectures.

However, low-level concurrency primitives like threads, semaphores, mutexes, or locks are considered error-prone, specifically for complex systems [Lee 2006]. This problem is addressed by Sutter's article "*The free lunch is over*" [2005], which started a discussion about the future of software development in the wake of Chip Multi Processors (CMPs) and concurrent programming. As a result, scalable RIS architectures which utilize concurrency while avoiding common pitfalls of parallel programming are an important research topic.

Profiling and benchmarking concurrent architectures requires additional care. Usually, benchmarking is performed by creating a benchmark scenario, adding measurement code to the system, and performing the measurement. This simple approach is not easily applicable to RIS development. First, too often, artificial small problems are chosen as concurrency benchmarks which can be parallelized easily due to their data-parallel aspects and which can not be compared to the parallelization of full-blown real world applications [Best et al. 2009]. Second, the instrumentation for benchmarking usually has to be coded into the system before execution. When the benchmark is completed, the instrumentation code is removed from the system. This has two disadvantages:

1. **Cost:** Design and implementation of the instrumentation has to be performed by the developers.
2. **Replicability:** Individual solutions and later removal of the instrumentation code hinders replicability.

Automated instrumentation using a profiling tool seems necessary. It frees up developers from designing a specialized instrumentation and it can make benchmarks more reproducible. Several of such tools exist but their applicability to modern RIS architectures is limited. These architectures usually incorporate some sort of event system as a central execution concept [Steed 2008b]. Event systems are highly beneficial to design modular systems and they additionally provide ideal target primitives for concurrent architectures, specifically if events are realized by–potentially asynchronous–message passing architectures. Now, standard profiling tools are agnostic to such middleware concurrency primitives and hence are largely useless here. To tackle this problem this article introduces

1. a set of metrics to compare the performance of event-based and message-passing-based asynchronous RISs.
2. an implementation of an instrumentation tool to apply the defined metrics and to run benchmarks.

Profiling and benchmarking RISs is an important increment towards sound and comparable user studies. Using monolithic closed source systems, developers can only measure end-to-end performance, e.g., frame rates or latency, and report the results. They can not control them. Causal relations between end-to-end measures and intrinsic system aspects can not be derived and negative impacts on user performance hence can not be reduced or prevented.

In Section 2, we give a short review of concurrency in RISs, performance metrics, and profiling tools. In Section 3, a set of metrics for message-passing-based asynchronous RISs is described. A prototypical implementation is outlined in Section 4. A case study using this implementation is described in Section 5. Section 6 presents the results and discusses future work.

2 Related Work

Coarse-grained concurrency schemes have been added to existing systems like Lightning [Bues et al. 2008], OpenMASK [Margery et al. 2002], ViSTA [Assenmacher and Kuhlen 2008], DLoVe [Deligiannidis 2000], Avango [Tramberend 2003] for some time now. Often, message-passing is used for clustering [Tramberend 1999; Schröder et al. 2010; Deligiannidis 2000; Allard et al. 2002].

Recent approaches support finer-grained concurrency as a key feature, e.g., FlowVR [Lesage and Raffin 2008] and Simulator X [Latoschik and Tramberend 2011]. They often realize concurrency by message passing to avoid several of the problems described by Lee [2006], e.g., dead locks and access violations. Still, concurrency concepts may vary drastically, e.g., while Simulator X

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

uses Hewitt's actor model [Hewitt et al. 1973], FlowVR is based on a data flow network. A side effect of message passing often introduces non-blocking behavior, a beneficial feature since it allows for asynchronous execution exploiting the power of CMPs. Here, required synchronization primitives, e.g., to render consistent world states, have to be provided explicitly.

Often, the available RIS middleware is distributed as a core systems with some additional applications [ViSTA 2013; Simulator X 2014]. Some also provide content editors, but to our knowledge no system provides a dedicated tool for performance measurement and profiling, a central feature of SGI's Performer [Rohlf and Helman 1994] software, which is not available any more.

2.1 Profiling and Monitoring in General

Profiling tools such as VTune [2013], YourKit [2014], and VisualVM [2014] typically provide features to measure time spent inside functions, to monitor memory (de)allocation, to trace call graphs, and to visualize these. Most profiling tools nowadays monitor concurrent aspects, i.e., degree of parallelism, congestion on synchronization primitives, and deadlocks and race conditions. These tools provide fine-grained and detailed information. They assume that threads and synchronization primitives are used directly to utilize parallelism. They are agnostic to middleware architecture concepts like events and messages and only measure the low-level primitives used for their implementation. Most importantly, latency effects caused by the asynchronous nature of events and messages are hardly traceable with a classic profiling tool and event and message cascades are only implicitly monitored by the function-based call graphs.

An example of a dedicated tool to monitor and profile a message-passing-based application is the Typesafe Console for the Akka framework [Typesafe Inc. 2014]. Typesafe Console runs as a server that collects data from a running application. The collected information is presented via a web interface. The provided metrics include information about latency and throughput in addition to Akka specific aspects. Unfortunately, none of the metrics provide information about consistency. Consistency describes the integrity of data exchanged between concurrent execution paths. A typical example in the RIS domain is the data exchange between simulation components and output render channels, which should only render consistent world states. It is debatable if consistency is really something that should be checked by a profiler. Detecting race conditions that cause inconsistency can either be a feature of a profiler like YourKit or a debugger such as Intel's Parallel Inspector.

Furthermore, latency measurement is not configured on the fly, but needs explicit instrumentation, the process of injecting the monitoring and measurement code into the system. Instrumentation typically increases the execution time of an application. The hard part is to find an approach that provides all the required measures and to calculate the metrics correctly while reducing the negative performance impact of the instrumentation itself. Here, many profiling tools do not collect every event, but instead use a sampling approach.

To summarize, the focus of classic profiling tools is too low-level and too fine-grained while ignoring middleware aspects like events and messages. Specific tools to monitor or profile message-passing-based applications overcome these problems. They provide an ideal starting point and first useful metrics to profile systems without measuring information about implementations-details.

2.2 Common metrics

Software metrics are objective, reproducible, and quantifiable measurements of some property of a piece of software or its specification. According to Hollingsworth et. al. [1995], they should help programmers to reduce execution time and to find errors and bugs. Typically, a set of metrics is required to describe the performance of an application comprehensively. According to Hu and Gorton [1997], a set of metrics should fulfill the following three requirements:

MR1 *Low variability*, which means that the value of the metric changes slowly according to the described aspect.

MR2 *Non-redundancy*, which means that metrics in the set of metrics do not overlap.

MR3 *Completeness*, which means that the set of metrics captures the whole system.

No standard set of metrics exists that describes every program [Hu and Gorton 1997], and sequential metrics cannot be simply extended to parallel programs [Hollingsworth et al. 1995]. As a staring point we identified the following five common metrics from the literature (scientific publications, benchmark suites, profiling tools, books on computer graphics and game engine development, and computer game magazines):

M1 Speedup

M2 Efficiency

M3 Degree of parallelism

M4 Frames per second

M5 Latency

M1 & M2: Speedup and efficiency

With respect to parallel systems, *speedup* compares the total execution time of a parallel algorithm or program with the total execution time of a serial implementation that solves the same problem. Speedup is often used to measure parallelization approaches of single aspects within a RIS, as in [Sigitov et al. 2013]. The *efficiency* is calculated by dividing the speedup by the number of processors.

M3: Degree of Parallelism

The *Degree of Parallelism* (DOP) shows how the parallelism has changed over execution time [Hollingsworth et al. 1995]. First, a *Program Activity Graph* (PAG) is created. In this graph, nodes represent significant events in the program's execution, while edges represent execution or wait times. Significant events are calls of functions, returns from functions, and locking or unlocking of synchronization primitives. Second, for each node the number of currently running parallel executions is determined and saved. As a result, the DOP of every moment of execution is available.

M4: Frames per Second

A very common metric for RISs is *frames per second* (fps), also called frame rate [Bierbaum 2000; Deligiannidis 2000]. This measures the overall frequency of an application on the hardware on which it is executed. The software is benchmarked by changing the application and running it on the same hardware. The hardware is benchmarked by executing the same application on different computers. 3DMARK by FUTUREMARK [2014] is a common commercial benchmark used by many computer game magazines to benchmark hardware. It also measures the fps achieved while rendering a test scene.

M5: Latency

Latency describes the delay between the start of an execution and its result. It is a very important metric of a RIS, especially for VR applications [Steed 2008a; Bierbaum 2000; Deligiannidis 2000]. While a high latency in computer games may just be annoying, it can result in simulator sickness in VR, or can have serious consequences in a medical application during surgery. It is necessary to distinguish between two types of latency. The *outer latency* measures a system like a black box, including software and hardware. It is not measured by the software itself, but by external methods as in [Steed 2008a] or device-specific hardware, like the Oculus VR Latency Tester [Oculus VR 2014]. The *inner latency* is measured by the–potentially instrumented–software itself. It provides information about the internal execution behavior. Inner latencies often cause outer latencies. Hence it is highly beneficial to measure inner latencies as a stating point for optimizations.

2.2.1 Discussion

Care has to be taken when adopting the metrics M1–M5 to RISs. As a special application field, RISs have their own requirements [Waldo 2008] that differ from general concurrency scenarios [Abdelkhalek and Bilas 2004]. With respect to the latter, the used metrics M1 and M2 (speedup and efficiency) typically assume that the workload is known at the beginning of the execution. Furthermore, it is assumed that a program starts, calculates a result, and then terminates. RISs typically run in interactive loops until the user terminates them. There is no final state or final result. Because of the interactive nature of RISs, workload might change unpredictably due to the simulation state and the user interaction, e.g., a changing field of view. Hence, metrics M1 and M2 largely loose their applicability and will not be incorporated later.

Asynchronous behavior exhibits a second pitfall when it comes to the omnipresent metric M4 (fps). This metric only measures the system performance for synchronous RISs, which are not frame-locked to any output device. In an ideal asynchronous system, the rendering component always renders the last known world state, regardless of how much time other components need, potentially rendering the same frame multiple times. Hence, the metric M4 only measures the rendering component but not the whole system. The metric loses its expressiveness the more asynchronous the system becomes. It can (and is later on) still be used when complemented by additional metrics designed to capture the frequency of looping execution parts in general.

3 Target Metrics

This section describes a set of metrics designed to work for most message-passing-based systems and many event-based architectures while fulfilling the three requirements MR1–MR3. We begin by introducing a formal language to precisely define the metrics followed by a detailed description of adapted and concretized metrics M3–M5, which are complemented by additional metrics M6–M11 that target specific aspects of events and messages. Note that the sequence of the introduction of the different metrics differs from section 2.2 to account for logical dependencies of certain metrics. The formalized language is inspired by Lamport [1978] but many operators have a slightly different meaning. There already exist several formalized languages for specific message-passing-based paradigms like CSP [Roscoe et al. 1997] or Hewitt's Actor model [Agha 1985]. However, CSP strongly focusses in modelling the state transitions of parallel processes, but does not allow to model the communication in a way it is useful for the presentation of the metrics in this article. The formalizms presented by Agha [1985] more focusses in describing the internal behavior of an actor. Hence, we decided to develope a new lightweight formalizm to present the metrics. While the formalized language uses the term *messages*, it is also applicable to events, specifically if the events are executed asynchronously.

3.1 Processes, Messages, and sending of messages

Let A and B be *processes*. We use the term process for code that is executed at least concurrently to other processes and communicates via messages with them. The term should not be confused with the term used in concrete implementations like processes in Windows or Linux or the term "thread". Both terms contain implementation details that are irrelevant for the descriptions of the metrics.

Messages contain information that is communicated from one process to another. Furthermore, messages can trigger a process to perform a calculation. The fact that A sends message m to B is written as $A \overset{m}{\to} B$. Upon $A \overset{m}{\to} B$ we assume that m is eventually processed by B. Furthermore, we assume that every message can only be sent once, so a message only has one sender and one receiver. Sending of multiple messages to another process(es) is written as $A \overset{m}{\to} B \Rightarrow A \overset{n}{\to} B$. This means that message m was sent before message n. We also assume that messages are processed in the same order they were sent.

For each process, two sets exist that keep a record about sent messages A_s and received messages A_r. The upper case letter of the set corresponds to the identifier of the process. Based on $A \overset{m}{\to} B$ the message m is in A_s and in B_r (the set keeps record of all messages processed by B).

3.2 Operations

Three timestamps exist for every message: when the message was sent $t_s(m)$, when processing began $t_b(m)$, and when it ended $t_e(m)$. The type α of a message m determined by the function $T(m)$.

Paths of information (and hence execution) can be written by $A \overset{m}{\to} B \overset{n}{\to} C \overset{o}{\to} D$. The total time span the information travels through the system can be written as $|A \overset{m}{\to} B \overset{n}{\to} C \overset{o}{\to} D| = t_e(o) - t_s(m)$, the time span between message m has been sent and message o has been processed.

For many metrics, only a subset that contains messages of a specific type is required. The set A_r^α is a subset of A_r that contains messages of type α.

$$A_r^\alpha = \{m \in A_r | T(m) = \alpha\} \qquad (1)$$

For other metrics, messages that are processed in a specific time span are required. Let S be a set that contains messages, b be the beginning of the time span, and e the end of the time span. The function $\Delta(S, b, e)$ returns a set that contains all messages in S that were processed between b and e.

$$\Delta(S, b, e) = \{m \in S | t_b(m) \geq b \wedge t_e(m) \leq e\} \qquad (2)$$

M6: Consistency

A metric that measures the consistency is not used to analyze the performance, but to find the reasons for bugs and unusual behavior of an application. In this metric, the communication between two processes $A \rightarrow B$ is analyzed. We assume that the world state in B is updated from the data of A by one or more messages called *update messages*.The first of the update message is a, while we call the last one z. The update is written as:

$$A \overset{a}{\rightarrow} B \Rightarrow A \overset{b}{\rightarrow} B \Rightarrow \ldots \Rightarrow A \overset{z}{\rightarrow} B \tag{3}$$

Usually, a process should work on a consistent world state. This is true if it has processed all update messages from another process prior to treat the message that triggers the simulation. When the type of the message that triggers the simulation is α, the target process worked on a consistent world state for this update when the following equation is true:

$$\{m \in B_r^{\alpha} | t_b(a) < t_b(m) < t_b(z)\} = \emptyset \tag{4}$$

This metric states in percentages, how often this equation is true.

M3: Degree of Parallelism

The Degree of Parallelism provides detailed information about how parallelism changes while the application is running. As stated previously, the DOP is computed by analyzing a PAG. To compute the DOP of a message-passing-based environment, the processed messages of all processes are collected in one set.

$$\Sigma_r = A_r \cup B_r \cup \cdots \cup N_r \tag{5}$$

The DOP is the number of messages processed concurrently at a given point in time. The DOP changes when the processing of a message starts or ends. For the calculation of the DOP, two tuples are generated for each message, one that contains the time stamp of the beginning ($\nearrow$) and the end ($\searrow$) of the processing.

$$\Pi = \{(t, \nearrow) | t = t_b(m) \wedge m \in \Sigma_r\} \cup \{(t, \searrow) | t = t_e(m) \wedge m \in \Sigma_r\} \tag{6}$$

Let us assume that $\Pi(i)$ provides access to each tuple in Π by the index i. Furthermore, assume that $\Pi_1(i)$ is the time stamp while $\Pi_2(i)$ is the symbol that determines if this tuple is the beginning or the end of a message processing. The tuples are accessed in an ordered way, sorted by the time stamp, so that $\Pi_1(i) \leq \Pi_1(i+1)$. Upon this information, the DOP can be calculated for any point in time t.

$$d(t) = \begin{cases} 0, & \text{if } \Pi_1(1) < t \\ e(t, 2, 1), & \text{if } \Pi_1(1) \geq t \end{cases} \tag{7}$$

$$e(t, i, d) = \begin{cases} d, & \text{if } \Pi_1(i) < t \\ e(t, i+1, d+1), & \text{if } \Pi_2(i) = \nearrow \\ e(t, i+1, d-1), & \text{if } \Pi_2(i) = \searrow \end{cases} \tag{8}$$

3.3 Counting metrics

Counting metrics count the number of messages and may set it into relation to other values.

M7: Number of messages per type

To analyze the communication, the types and number of sent and processed messages are analyzed. With this metric a developer can determine if the communication occurs in the way he/she expected. To count the number of messages, the type of the messages α and a time span defined by the beginning b and the end e are required. With these three parameters the metric for process A is defined as:

$$m_A(\alpha, b, e) = |\Delta(A_r^{\alpha}, b, e)| \tag{9}$$

M8: Messages per second

The messages per second show the overall throughput of the system. This metric is computed by counting all processed messages within the time span between b and e and dividing the number of messages by the time span. The messages per second of the whole application is calculated by the following equation:

$$s(b, e) = \frac{|\Delta(\Sigma_r, b, e)|}{e - b} \tag{10}$$

M4: Frequency of a process (former: fps)

While the fps lose their expressiveness in a message-passing-based and asynchronous RIS as an overall metric for the whole system, the frequency of a process that represents a component is useful information to estimate the overall performance. Usually, a RIS is built out of several components such as rendering, physics, and AI. In a message-passing-based and asynchronous RIS, every component runs within at least one process. Typically, each type of component has its own frequency, such as 60 Hz for rendering, 120 Hz for physics simulation, and 1kHz for haptics devices. Measuring frequency provides a quick overview if the desired frequency is achieved. Even when a component uses multiple processes to parallelize the computation, they usually get synchronized on one process that represents the component to achieve a consistent result of the component's task. However, not all components work in a loop. Some are reactive only, answering on request. In this case no frequency can be calculated, but *latency* is the right metric to measure the performance.

The frequency of the process A for the message type α in the time span between b and e is calculated by:

$$f_A^{\alpha}(b, e) = \frac{m_A(\alpha, b, e)}{e - b} \tag{11}$$

3.4 Timing metrics

Timing metrics use some measured time related to a message and may set it into relation to other values.

M9: Processing time per message type

The processing time per message is an important metric to identify critical parts of an application. This metric is comparable to the computation time per function, usually measured by classic profiling tools. The processing time is the time span $t_e(m) - t_b(m)$. To calculate the average time in the time span between b and e, let's assume that the function $P(A_r)$ calculates the summed up processing time of all messages within the set. The average time that the processing of a message of the type α takes is defined as:

$$p_A(\alpha, b, e) = \frac{P(\Delta(A_r^\alpha, b, e))}{m_A(\alpha, b, e)} \tag{12}$$

M10: Ratio between simulation and overhead

Message-passing-based asynchronous RISs do have a larger overhead than synchronous RISs by design. To calculate this metric, one type α needs to be defined as the type of the message that triggers the simulation. Next, a set that contains all messages processed by A without the trigger messages needs to be generated. This set is defined as $A_r^r = A_r \setminus A_r^\alpha$. Then, the ratio r_A^α can be calculated by the following equation:

$$r_A^\alpha(b, e) = \frac{P(\Delta(A_r^\alpha, b, e))}{P(\Delta(A_r^r, b, e))} \tag{13}$$

M5: Latency

The latency metric is defined as:

$$|A \overset{m}{\to} B \overset{n}{\to} C \overset{o}{\to} D| = t_e(o) - t(m) \tag{14}$$

However, $|A \overset{m}{\to} B \overset{n}{\to} C \overset{o}{\to} D|$ is not equal to $|A \overset{m}{\to} B| + |B \overset{n}{\to} C| + |C \overset{o}{\to} D|$. How the overall latency is distributed along the execution path highly depends on the implementation of the system. The communication structure of the actual system needs to be analyzed carefully. A possible implementation of this metric is explained in Section 4.4.

M11: Time a message waits before it is processed

This metric helps to identify overloaded processes. The waiting time is calculated by subtracting the time stamp when the message has been sent and the time stamp when the processing has started. When a message is sent $A \overset{m}{\to} B$, the time a message waits is $t_b(m) - t_s(m)$. Let's assume that $W(A_r)$ calculates and sums up the time span $t_b(m) - t_s(m)$ for all messages in A_r. Then, the average time a message waits before it is processed for a time span between b and e is calculated by:

$$w_A(b, e) = \frac{W(\Delta(A_r, b, e))}{|\Delta(A_r, b, e)|} \tag{15}$$

3.5 Analyzing Minimum, Maximum, Average, and Variance

Several of the metrics M3–M11 calculate values based on an interval between b and e. This aggregated information provides a basis for further analysis approaches. The whole execution time of the application can be split into small time slices, and the metrics are calculated for each of the time slices. Afterward, the minimum, maximum, and variance can be calculated. Minimum and maximum usually represent best cases and worst cases. Whether the minimum is the best case or the worst case depends on the metric. For all metrics, the variance should be low.

4 Implementation

None of the tools presented in Section 2.1 provide the metrics M3–M11 defined in Section 3. Therefore, an implementation of a specific profiling tool is necessary.

In this section, a prototypical implementation of our approach is described. First, we define requirements for the implementation. Next, we describe the Simulator X framework and the underlying actor model implementation Akka. Afterward, we outline the instrumentation. Next, the implementation of the metric M5 (latency) is explained. Finally, we give an overview about the implementation of the data viewer.

4.1 Requirements

For the overall implementation, we define the following two requirements:

IR1 *Completeness:* The implementation has to collect data to calculate the metrics M3–M11 and has to visualize them appropiately.

IR2 *Reusability:* As many aspects of the implementation as possible should be reusable to implement profiling tools for other RISs.

The requirement IR2 implies to split the implementation into the instrumentation and a data viewer. While the instrumentation has to be tailored for a specific RIS, a tool that calculates the metrics and visualizes the results can be independent of the concrete RIS. For the instrumentation we define the following two requirements:

IR3 *Generality:* Instrument generally instead of specifically. Instrument the message-passing implementation rather than the RIS implementation, and instrument the RIS implementation rather than the application implementation.

IR4 *Minimal invasiveness:* Target the instrumentation to produces a low overhead.

To ensure the reusability of the data viewer, we define the following three requirements:

IR5 *Portability:* Implement the data viewer independent from a specific operating system and hardware architecture.

IR6 *Extensibility:* Support software hooks and stubs for further extensions, e.g., new metrics.

IR7 *Modularity:* Choose a modular design that provides reusability.

4.2 The Simulator X Framework

The metrics M3–M11 described in this paper have been implemented into a profiling tool for the Simulator X Framework [Latoschik and Tramberend 2011], an experimental RIS middleware based on the Scala programming language. Scala is a modern multi-paradigm language that contains object-oriented and functional language features [Odersky 2011]. Scala uses the actor model as the fundamental concurrency model and it uses the Akka actor implementation since version 2.10. The central programming concepts of Simulator X—components, entities, and state-variables—and their integration with the actor model are briefly described here. Furthermore, the underlying actor model implementation Akka is described.

Components provide essential services for RIS applications within Simulator X, e.g., rendering, physics simulation, or artificial intelligence, etc. Components are high-level functional building blocks providing different aspects of a simulation. Components are implemented as actors providing coarse-grained concurrency.

Entities encapsulate visible as well as invisible simulation objects. They aggregate associated state variables. An entity can have state variables for specific attributes, e.g. for 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.

State variables provide the dynamic 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. Since state variables are managed by actors, they provide the fine-grained concurrency model.

Actors are always created within an actor system. According to Typesafe Inc. [2014] an actor system allocates one or more threads for actors and manages the scheduling of actors and dispatching of messages. Each actor encapsulates its own state and defines its behavior in form of a handler function that processes messages according to the type of the message. To protect the internal state of the actor, the actual instance of an actor is not accessible. Instead, an actor reference (ActorRef) is used to send messages by using the !-operator.

When a message is sent to an actor, the message is enqueued to the mailbox of the actor. The dispatcher of the actor system knows the mailbox of each actor. When computation time is available, the dispatcher calls a function of the mailbox. Within this function, a message is taken from the mailbox and passed to the behavior function of the actor.

4.3 Instrumentation

To collect data about sent messages the !-operator of the ActorRef is instrumented. To collect information about the processing of messages the mailbox of an actor is instrumented. To measure the processing time, the time stamp before a message is dispatched to the behavior function is taken. Another time stamp is taken when the behavior function has finished. The instrumentation takes time stamps in nanosecond resolutions, because a lower resolution is too coarse-grained to get reasonable information about latency and consistency. The collected time stamps are written to a csv-file when the application is closed. Because the implementation only instruments the message-passing implementation, it fulfills the requirement IR3. Furthermore, it fulfills the requirement IR2, because is can be used for any RIS implemented on top of Akka.

We benchmarked the overhead caused by the instrumentation by sending 1,000,000 messages between two actors and measuring the overall time it takes to send, receive, and process the messages. The uninstrumented version takes about 1s while the instrumented one takes about 4.5s. This overhead is of course noticeable, a qualitative comparison to standard profiling tools reveals this overhead to be fairly justifiable. Still, further benchmarks should compare the instrumentation to the implementations of other profilers on a quantitative basis to check for requirement IR4.

4.4 Implementation of the Latency Metric (M5)

As described in Section 3.4, the concrete system needs to be analyzed carefully to provide a useful implementation of M5. When some new information is generated within the system, it is communicated to other actors and processed there. For example, a transformation of a simulated object is generated by the physics component and sent to the graphics component to render the correct position and orientation of the object in the rendered frame. Each component is implemented in its own actor. This is the base scenario for the following analysis.

Two different message types are essential for the example base scenario. The processing of *simulation loop messages* performs the simulation of the component, like physics calculation or rendering. *Update messages* communicate the information from one component to another. The physics component generates new transformations for simulated objects while it processes a simulation loop message. The calculated transformations are written to state variables. This results in sent messages that contain the transformations to the rendering component. The rendering component processes the messages and updates its scene representation. Eventually, the rendering process receives a simulation loop message and renders the new frame while processing it.

Given:

- X: A process that triggers other processes by sending simulation loop messages.
- A: The physics component.
- B: The rendering component.
- a: The first update message.
- z: The last update message.
- s_x: Simulation loop messages.

The aforementioned scenario can then be written as:

$$X \xrightarrow{s_1} A \Rightarrow A \xrightarrow{a} B \Rightarrow A \xrightarrow{b} B \Rightarrow \ldots \Rightarrow A \xrightarrow{z} B \Rightarrow X \xrightarrow{s_2} B \tag{16}$$

In context of inner latency, the earliest point in time when the new transformation could be known is $t_b(s_1)$, while the latest point in time the new transformation is available at the output channel for the user is $t_e(s_2)$. The overall inner latency $t_e(s_2) - t_b(s_1)$ helps to identify that there *is* a high latency, but not to identify *where* it comes from. Therefore, the overall inner latency needs to be split into several intervals.

We recognized, that in Simulator X update messages are usally sent while simulation loop messages are processed. Therefore, not the time span $t_e(s_1) - t_b(s_1)$ is interesting, but the time until the update message is sent, that is $t_s(a) - t_b(s_1)$. Next, the update message waits in the mailbox of the receiver, what is $t_b(a) - t_s(a)$. Afterwards, the rendering component updates its scene graph, hence the information waits until the next simulation loop message is processed by the rendering component, that is $t_b(s_2) - t_b(a)$. Finally, the rendering component processes the simulation loop message and finishes rendering $t_e(s_2) - t_b(s_2)$. The sum of all of these time spans equals the overall inner latency, but the time spans help to identify the reason of the latency.

Beside the time spans of the example, we identified some more time spans for the case an actor does not have a simulation loop message. All identified time spans are in Table 1.

Using the IDs of the time spans of Table 1, the example can be visualized as in Figure 1.

4.5 Data viewer

The data viewer has been implemented using the NetBeans RCP Platform as a solid foundation for multi-window applications. NetBeans RCP applications run on all major desktop operating sys-

Table 1: *Identified time spans for metric M5 in the context of Simulator X.* ***s*** *represents simulation loop messages, while* ***u*** *represents update messages.*

ID	Description	Formalized	Possible Successor
1	Begin of the processing of the simulation loop message until the update message is in mailbox	$t_s(u) - t_b(s)$	2
2	Update message waits in mailbox	$t_b(u) - t_s(u)$	3, 4, or 5
3	Begin of processing of the update message until the next update message is in mailbox	$t_s(u_2) - t_b(u_1)$	2
4	End of last time span until final update message processed completely	$t_e(u) - t_b(u)$	—
5	Begin of processing of the update message to begin of processing of the simulation loop message	$t_b(s) - t_b(u)$	1 or 6
6	End of last time span until final simulation loop messaage processed completely	$t_e(s) - t_b(s)$	—

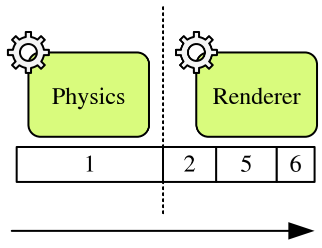


Figure 1: *An example how the flow of information can be modeled by the states illustrated of Table 1.*

tems, hence IR5 is fulfilled. Furthermore, the NetBeans RCP plug-in structure provides future extension capabilities, hence IR6 is fulfilled. To use the data viewer for other RISs some plug-ins that are specific for Simulator X need to be replaced by tailored implementations. For example, the implementation of the latency metric is implemented by a Simulator X-specific plug-in. This plug-in just needs to be replaced by another latency metric implementation for the specific RIS, hence IR7 is fulfilled. The data viewer visualizes the data captured by the instrumentation. It implements all the metrics M3–M11 and completes IR1.

Figure 2a illustrates a screenshot of a latency report for the latency between the physics component and the renderer in a demo application of Simulator X. The "General" group shows the minimum, maximum, average, and median latency. In the "Graphical" group the latency is plotted over time. In the "Path" group the time spans of the path is shown in a table.

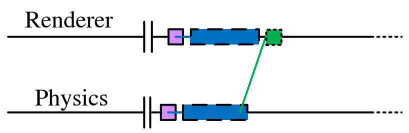
5 Case Study: Profile an example application of Simulator X

We used the profiler to track down a high latency in an example application shipped with Simulator X. The example application represents a typical RIS configuration. It consists of three components: A physics component, a rendering component, and a sound component. The application simulates a basic scene where a ball jumps on a table. When the ball collides with the table, a sound is played. The jumping ball and the table is simulated by the physics component and rendered by the rendering component.

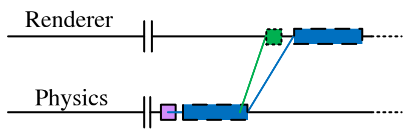
We used the latency metric (M5) to analyze how the latency is distributed along the path of execution and flow of information in the application (fig. 2a). We identified a noticeable latency between the processing of the update message by the the rendering actor and the start of the next simulation loop (time span 5 of Table 1).

The data viewer revealed this behavior. It visualized the communication between actors and processing of messages in Figure 3. Specifically, Figure 3a illustrates the faulty synchronization between the physics and the rendering component. The purple rectangle with the solid outline is the processing of a trigger message sent by Akka. The blue rectangle with the dashed outline shows when the simulation loop message was processed. The green rectangle with the dotted outline shows the processing of the update message. Colored lines show when the message was sent. According to Mönkkönen [2006], the components run independently from each other. Because Akka triggers both components at the same time, the update message is processed after the renderer renders the next frame. As a result, the update is rendered at the next frame, what increases latency.

We changed the application, so that the rendering actor is not triggered by Akka with a fixed frequency, but is triggered by the physics component, as shown in Figure 3b. Coupling the renderer to the physics component reduced the overall latency as shown in Figure 2b, but also decreased the parallelism.



(a) *According to Mönkkönen [2006] the components run independently from each other, each on a fixed frequency. This increases parallelism, but introduces latency.*



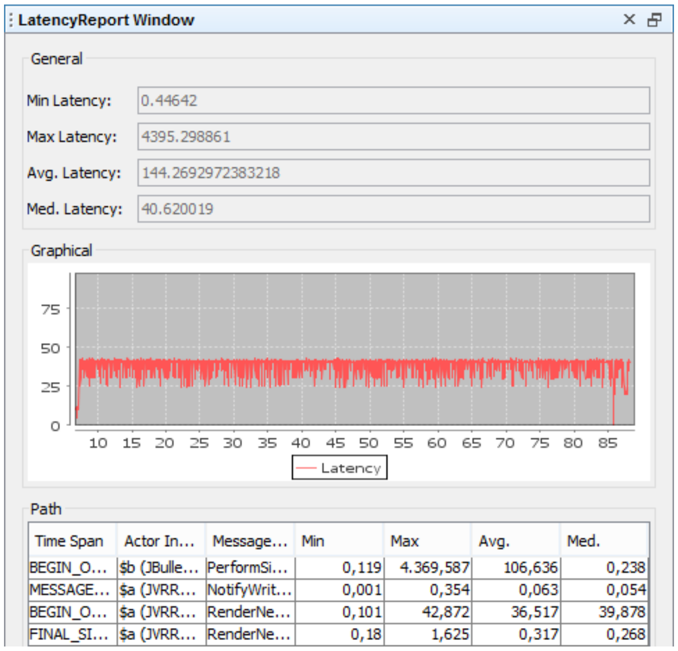
(b) *The physics component triggers the renderer. This reduces latency, but also reduces the parallelism.*

Figure 3: *Visualized message sending and processing of two components.*

6 Conclusion and Future Work

This paper introduced a set of metrics (M3–M11) to profile and benchmark decentralized and asynchronous RISs based on events or message passing. The set of metrics fulfills MR1 by having a low variability, MR2 by being non-redundant, and partly fulfills MR3 by being *nearly* complete. The set of metrics is fairly complete because the consistency metric (M6) relies on components as independent actors. Future extensions of M6 have to cope with derived patterns where a component is implemented by multiple actors. We also want to extend the latency metric (M5). Currently, only a single path can be described and analyzed. Analyzing existing applications yields the result that patterns exist where execution is performed by multiple paths through the application.

All the metrics can be calculated upon data collected by an instrumentation that only instruments the underlying message-passing implementation, but not the RIS itself or the application. As a result, the instrumentation can be reused for any message-passing-based RIS implemented on top of Akka. Furthermore, we implemented a data viewer for the measured data. The overhead of the



Min Latency: 0.44642

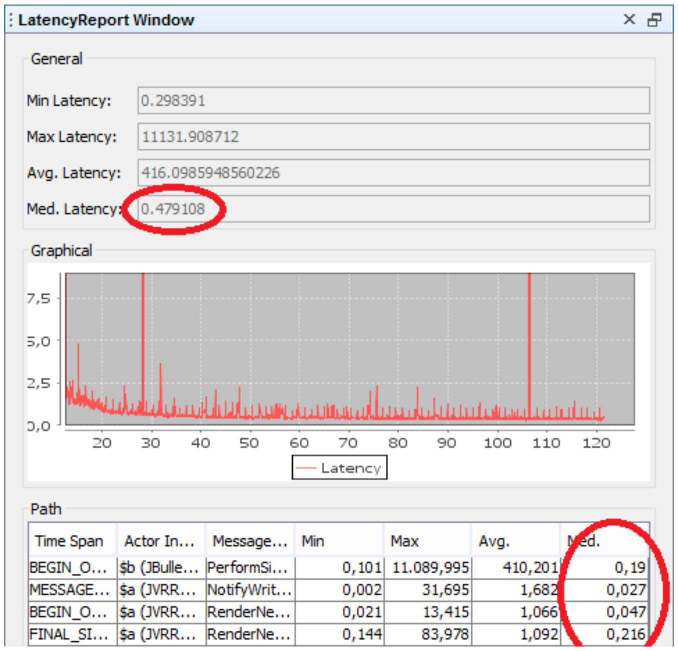
Max Latency: 4395.298861

Avg. Latency: 144.2692972383218

Med. Latency: 40.620019

Time Span	Actor In...	Message...	Min	Max	Avg.	Med.
BEGIN_O...	$b (JBulle...	PerformSi...	0,119	4.369,587	106,636	0,238
MESSAGE...	$a (JVRR...	NotifyWrit...	0,001	0,354	0,063	0,054
BEGIN_O...	$a (JVRR...	RenderNe...	0,101	42,872	36,517	39,878
FINAL_SI...	$a (JVRR...	RenderNe...	0,18	1,625	0,317	0,268

(a) *A screenshot of a latency report. Because the application run for a short period, the average latency is highly influenced by the maximium latency from the start up phase of the application.*



Min Latency: 0.298391

Max Latency: 11131.908712

Avg. Latency: 416.0985948560226

Med. Latency: 0.479108

Time Span	Actor In...	Message...	Min	Max	Avg.	Med.
BEGIN_O...	$b (JBulle...	PerformSi...	0,101	11.089,995	410,201	0,19
MESSAGE...	$a (JVRR...	NotifyWrit...	0,002	31,695	1,682	0,027
BEGIN_O...	$a (JVRR...	RenderNe...	0,021	13,415	1,066	0,047
FINAL_SI...	$a (JVRR...	RenderNe...	0,144	83,978	1,092	0,216

(b) *A screenshot of the same application as in Figure 2a, but with changed trigger order. The latency has been reduced drastically by using the triggering order visualized in Figure 3b.*

Figure 2: *Screenshots of the latency report of the data viewer. The latency report visualizes the results of the latency metric.*

instrumentation has been measured, but further analysis is necessary to qualify the overhead.

A case study already proved the usefulness of the metrics, i.e., the latency metric and the implementation of the profiling tool. The identified latency was not caused by a complex computation or a saturated actor, but by the scheduling of the actors and the communication sequence. These kind of problems are hard to be identified using classic profiling tools, which emphasizes the importance of specialized metrics and tools for analyzing RISs. The latency was reduced by changing the trigger sequence of the components. The case study implies that a correlation between latency and parallelism exists. A configuration layer to configure the trigger sequence and define synchronization seems favorable and is a future goal.

To integrate our approach into other systems, the major part is to develop an instrumentation to collect the required data. It should always be preferred to instrument the underlying message-passing implementation instead of instrumenting the RIS itself. Functions need to be identified where the required information is available and can be saved without a large overhead. The data viewer implementation can be reused. It only needs minor changes if the measured data is saved in the same or a similar file format.

The implementation does not replace classic profiling tools such as VTune, YourKit, and VisualVM. Instead, the Simulator X Profiler complements the profiling capabilities with high-level features. As a rule of thumb, for every low-level analysis a classic profiling is used. For every high-level task, the Simulator X Profiler is used. In Table 2, we list some typical tasks during application development and state which tool should be used for it.

Overall, profiling and benchmarking is an important factor of RIS development. Monolithic closed-source systems currently do not provide the necessary features nor do they allow source code access for the instrumentation. Hence, if profiling and optimization is required, e.g., to reduce unwanted or harmful perception artifacts in the user interface caused by deficient technical realizations, VR/AR developers need reliable metrics, profiling tools, and software systems which provide the necessary optimization access. The metrics and approach described here is a step towards this goal.

Table 2: *Tasks and Tools.*

	Classic Profiler	Simulator X Profiler
Identify *functions* that consume a lot of calculation power	✓	×
Identify *actors* that consume a lot of calculation power	×	✓
Profile memory behavior	✓	×
Analyze overall communication structure	×	✓
Analyze concurrency on low-level	✓	×
Analyze concurrency on high-level	×	✓
Check low-level consistency (find race conditions)	✓	×
Check high-level consistency	×	✓
Measure latency	×	✓
Identify congestion of actors	×	✓

Acknowledgements

The authors wish to thank Pedram Merrikhi.

References

ABDELKHALEK, A., AND BILAS, A. 2004. Parallelization and performance of interactive multiplayer game servers. In *Parallel* *and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 72–.

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

ALLARD, J., GOURANTON, V., LECOINTRE, L., MELIN, E., AND RAFFIN, B. 2002. Net juggler and softgenlock: Running vr juggler with active stereo and multiple displays on a commodity component cluster. In *Proceedings of IEEE Virtaul Reality Conference 2002*, 273–274.

ASSENMACHER, I., AND KUHLEN, T. 2008. The vista virtual reality toolkit. In Latoschik et al. [Latoschik et al. 2008], 23–26.

BEST, M. J., FEDOROVA, A., DICKIE, R., TAGLIASACCHI, A., COUTURE-BEIL, A., MUSTARD, C., MOTTISHAW, S., BROWN, A., HUANG, Z. F., XU, X., GHAZALI, N., AND BROWNSWORD, A. 2009. Searching for concurrent design patterns in video games. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Springer-Verlag, Berlin, Heidelberg, Euro-Par '09, 912–923.

BIERBAUM, A. D. 2000. *VR Juggler: A Virtual Platform for Virtual Reality Application Developement*. PhD thesis, Iowa State University.

BUES, M., GLEUE, T., AND BLACH, R. 2008. Lightning: Dataflow in motion. In Latoschik et al. [Latoschik et al. 2008], 7–11.

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

FUTUREMARK, 2014. `http://www.futuremark.com/benchmarks/3dmark`.

HEWITT, C., BISHOP, P., AND STEIGER, R. 1973. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

HOLLINGSWORTH, J. K., LUMPP, J., AND MILLER, B. P. 1995. Techniques for performance measurement of parallel programs. *Parallel Computers: Theory and Practice*.

HU, L., AND GORTON, I. 1997. *Performance evaluation for parallel systems: A survey*. Citeseer.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

LATOSCHIK, M. E., AND TRAMBEREND, H. 2011. Simulator X: A Scalable and Concurrent Software Platform for Intelligent Realtime Interactive Systems. In *Proceedings of the IEEE VR 2011*.

LATOSCHIK, M. E., REINERS, D., BLACH, R., FIGUEROA, P., AND DACHSELT, R., Eds. 2008. Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2008 workshop, Shaker Verlag.

LEE, E. A. 2006. The problem with threads. *Computer 39* (May), 33–42.

LESAGE, J.-D., AND RAFFIN, B. 2008. High performance interactive computing with flowvr. In Latoschik et al. [Latoschik et al. 2008], 13–16.

MARGERY, D., ARNALDI, B., CHAUFFAUT, A., DONIKIAN, S., AND DUVAL, T. 2002. Openmask: Multi-Threaded — Modular animation and simulation Kernel — Kit : a general introduction. In *VRIC 2002 Proceedings*, S. Richir, P. Richard, and B. Taravel, Eds., 101–110.

MÖNKKÖNEN, V., 2006. Multithreaded game engine architectures. WWW, Sep.

OCULUS VR, 2014. `https://www.oculusvr.com/order/latency-tester/`.

ODERSKY, M., 2011. The scala language specification version 2.9, may.

ROHLF, J., AND HELMAN, J. 1994. Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '94, 381–394.

ROSCOE, A. W., HOARE, C. A. R., AND BIRD, R. 1997. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

SCHRÖDER, D., WEFERS, F., PELZER, S., RAUSCH, D., VORLÄNDER, M., AND KUHLEN, T. 2010. Virtual reality system at rwth aachen university. In *Proceedings of the International Symposium on Room Acoustics (ISRA), Melbourne, Australia*.

SIGITOV, A., ROTH, T., MANNUSS, F., AND HINKENJANN, A. 2013. Drive: An example of distributed rendering in virtual environments. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2013 6th Workshop on*, 33–40.

SIMULATOR X, 2014. `https://github.com/simulator-x`.

STEED, A. 2008. A simple method for estimating the latency of interactive, real-time graphics simulations. In *Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology*, ACM, New York, NY, USA, VRST '08, 123–129.

STEED, A. 2008. Some useful abstractions for re-usable virtual environment platforms. In Latoschik et al. [Latoschik et al. 2008], 33–36.

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

TRAMBEREND, H. 1999. Avocado: a distributed virtual reality framework. In *Virtual Reality, 1999. Proceedings., IEEE*, 14 – 21.

TRAMBEREND, H. 2003. *Avocado : a Distributed Virtual Environment framework*. PhD thesis, Bielefeld University.

TYPESAFE INC., 2014. `http://doc.akka.io/docs/akka/2.3.2/general/actor-systems.html`.

VISTA, 2013. `http://sourceforge.net/projects/vistavrtoolkit/`.

VISUALVM, 2014. `http://visualvm.java.net/`.

VTUNE, 2013. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`.

WALDO, J. 2008. Scaling in games & virtual worlds. *Queue 6* (November), 10–16.

YOURKIT, 2014. `http://www.yourkit.com/features/`.