Björn Richerzhagen, Dominik Stingl, Julius Rückert, Ralf Steinmetz. *Simonstrator: Simulation and Prototyping Platform for Distributed Mobile Applications.* In Proc. EAI International Conference on Simulation Tools and Techniques (SIMUTOOLS), ACM, 2015. ISBN: 978-1-63190-079-2

Simonstrator: Simulation and Prototyping Platform for Distributed Mobile Applications

Björn Richerzhagen[*], Dominik Stingl[*], Julius Rückert[†], and Ralf Steinmetz[*]
[*] Multimedia Communications Lab (KOM), E-Mail: firstname.lastname@kom.tu-darmstadt.de
[†] Peer-to-Peer Systems Engineering Lab (PS), E-Mail: rueckert@ps.tu-darmstadt.de
Technische Universität Darmstadt, Germany

ABSTRACT

The increasing market penetration of mobile devices, such as smartphones and tablets, poses additional challenges on the design of distributed systems. Due to the heterogeneous environment consisting of both, mobile and fixed devices, a multitude of effects on different scales need to be considered. Microscopic effects, such as an individual user's interaction with the device, as well as macroscopic effects, such as scalability with the number of users have an impact on the system's performance. The combined evaluation of micro- and macroscopic effects requires both, simulations and prototypical deployments. Furthermore, insights obtained through prototypes during user studies can lead to refined protocols and algorithms, thereby contributing to the overall design process. To enable parallel assessment of micro- and macroscopic effects, we propose the Simonstrator platform, consisting of a lightweight framework for the development and instrumentation of distributed systems as well as runtime environments for (i) the interaction with common simulators, (ii) the deployment on testbeds, and (iii) Android devices. The platform is specifically targeted towards distributed systems for heterogeneous scenarios, considering mobile and fixed networks. We show sample simulations and prototypical deployments of two exemplary use cases: a live video streaming system and a middleware for augmented reality games, highlighting different evaluation goals and environments supported by the proposed Simonstrator platform.

1. INTRODUCTION

With the ubiquity of smartphones and tablets, an increasing number of distributed applications are running on mobile devices. On the one hand, this platform offers new opportunities due to the availability of sensor information and the tight coupling to the device user. On the other hand, the mobile environment poses additional challenges on the design of distributed systems. These challenges include a highly dynamic environment caused by user movement, unstable connectivity, as well as different usage patterns than known from fixed devices.

To design distributed applications that cope with these challenges, one has to evaluate the impact of the dynamic environment on the systems' performance. This has to be done on different scales, ranging from an individual user's interaction patterns with the device and other users to effects observed on a global scale, e.g., the scalability of the overall system. Depending on the scope of the evaluation, a multitude of simulators or emulators exist that address some of the effects by providing suitable models. However, it is well known that simulations can not capture the full range of effects caused by the environment [15]. Therefore, prototypical deployments and testbed evaluations need to augment the simulative study of a system [7]. Early prototypical insights can then be fed back into simulation models and algorithms to study effects at larger scale.

To this end, there have been proposals to enable simulation models to be deployed as prototypes [3, 5] or to integrate simulators into testbed deployments [8], thereby increasing the scale of the scenario. However, existing systems mostly neglect the peculiarities arising on mobile devices such as smartphones and tablets. This includes access to sensors and other device-specific features, which constitutes an important building block of today's distributed applications. Furthermore, the resulting system is often limited to one specific simulator. This, in turn, limits the range of effects that can be studied using validated models.

In this work, we present the Simonstrator framework that does not depend on a specific simulator or evaluation platform. Instead, systems designed using the proposed framework can be executed in any environment that provides the core functionality as required by the framework's programming interfaces. We provide and discuss the respective *runtime environments* comprising the Simonstrator platform, enabling systems built using our framework to run on (i) the network simulator OMNeT++ [17], (ii) the overlay simulator PeerfactSim.KOM [16], (iii) commodity PCs, and (iv) Android mobile devices. New runtime environments can be added easily by bridging the core functionality provided by the framework to the respective environment. Therefore, in contrast to existing solutions, applications implemented within our framework can be evaluated with a broad range of existing simulators and their models, as well as being deployed prototypical. Furthermore, applications can utilize the runtime's peculiarities, such as access to sensors on an Android device, through our component-based framework.

We study the applicability of the Simonstrator platform for two exemplary use cases: a distributed live video streaming system and augmented reality gaming. Insights from early prototypes of the augmented reality game are used to define important properties of the middleware design. Based on the players' movement and interaction patterns during a field study, the applicability of local ad hoc event dissemination is assessed. Subsequently, simulations enable fast assessment of a number of protocols suitable for the scenario. Due to the Simonstrator platform, these protocols can then be utilized in the prototype without any changes to the code. The insights gained from early prototypes stress the importance of combined prototypical and simulative evaluation.

The remainder of this paper is structured as follows. Section 2 details the Simonstrator platform, consisting of the Simonstrator framework and runtime environments. Two exemplary use cases for our platform are discussed in Section 3, highlighting its usage on mobile devices and simulators for two sample scenarios with distinct evaluation goals. Section 4 contains relevant related work. Section 5 concludes the paper and illustrates possible directions for future work.

2. PLATFORM ARCHITECTURE

The Simonstrator platform consists of two main building blocks, as illustrated in Figure 1. The *Simonstrator framework* represents the central building block of the platform (cf. Section 2.1), providing the base abstractions to be used by the distributed system. The framework is complemented by a set of *runtime environments* (cf. Section 2.2) that enable the deployment of applications in different environments. With our Simonstrator platform we follow the concept introduced by Galuba et al. with ProtoPeer [5] by providing time and network related functions through our framework. However, in contrast to ProtoPeer, the Simonstrator framework also includes access to sensors commonly used by today's distributed applications (e.g., location) as well as other communication methods (e.g., Bluetooth). Furthermore, it enables loose coupling of components, thereby allowing platform components to utilize protocol and application components and vice-versa.

The provided runtime environments enable code execution on Android devices, on common network simulators, and on a wide range of Java-based platforms. The architecture is realized with the Java programming language due to its high outreach on fixed and mobile (Android) platforms. In the following, the building blocks of the Simonstrator platform are discussed in detail.

2.1 The Simonstrator Framework

The Simonstrator framework can be divided into two conceptional layers, as indicated in Figure 1. The foundation for applications and systems built around the framework is the central scheduling abstraction. Here, representations for absolute and relative time are provided, enabling derived concepts such as events and operations. Furthermore, this layer includes instrumentation-related interfaces (c.f. Sec. 2.1.2) to capture relevant data from the conducted experiments for debugging or a subsequent evaluation. The application itself is composed out of different *components* utilizing these core abstractions. Components provide platform- or application-specific functionality. By combining multiple components

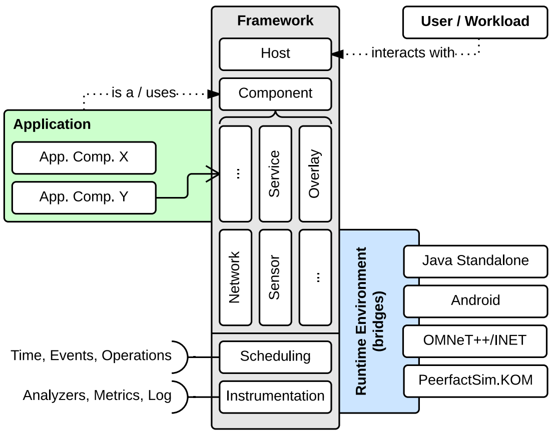


Figure 1: Overview of the Simonstrator platform consisting of the core framework and a set of runtime environments.

within a *host*, complex distributed applications are realized (c.f. Sec. 2.1.3). Please note that real-world code examples as well as additional documentation of the framework are provided in our public project repositories[1] and are omitted in this paper for brevity.

2.1.1 Time, Events, and Operations

To enable repeatable and deterministic executions of simulations, access to time as well as functionality related to threading and scheduling needs to be encapsulated. The framework allows the retrieval of the current time as well as relative time calculations required for delayed execution of tasks. Instead of creating and maintaining own threads for concurrent, delayed, or periodic execution of specific tasks, *events* and *operations* are created and scheduled through the framework. Using the aforementioned abstraction of time, an event is scheduled as shown in Listing 1. Once the event is due for execution, the provided event handler is notified. Depending on the runtime environment, deterministic execution order of events is guaranteed (cf. Sec. 2.2).

Listing 1: Event scheduling and timing abstractions.

```
EventHandler handler = new EventHandler() {
  public void eventOccurred(Object content,
      int type) {
    // [do something]
  }
}

Event.scheduleWithDelay(10*Time.SECOND,
    handler, null, 0);
```

While events are a reasonable abstraction for *one-shot* actions, most distributed systems rely on operations that represent an abstraction for an *action* $\rightarrow$ *reaction* type of interaction. Consider a simple *request and reply* case, where the sender of a message waits (i) for a reply to process it or (ii) reacts to a timeout beforehand. To maintain the state of the request (i.e., the timeout as well as the information contained in the request), isolation within a dedicated thread is desired. The concept of *operations* helps to realize

[1]`https://dev.kom.e-technik.tu-darmstadt.de/gitlab/maki/simonstrator-api`. Parts of the platform are not available publicly; please contact the authors for full access.

these types of scenarios by providing an intuitive, thread-like programming interface on top of *events*. In contrast to the abstraction achieved through single *events*, this enables a separation of concerns while simultaneously guaranteeing determinism and avoiding common concurrency issues. Last but not least, generators for random numbers are provided through the framework to enable deterministic seeding of experiments in simulation scenarios.

2.1.2 Instrumentation

Instrumentation constitutes an inevitable step during both, prototypical and simulative evaluation to output data for debugging and analysis. The Simonstrator framework provides access to platform-independent logging, using a simple wrapper that mimics Apache's popular Log4j framework. While logging is helpful during development and debugging, it is not suitable for full evaluation of a system. For this reason the Simonstrator framework offers two different concepts to ease the evaluation of an examined system: push-based *analyzers* and pull-based *metrics*.

Analyzers are directly triggered by the evaluated system and are application specific. Consider a distributed video streaming system: to assess its performance, one has to capture stalling events, where the user playback is interrupted due to empty buffers. A `PlaybackAnalyzer` interface would include methods such as `onStall` and `onResume` that are called by the streaming system as soon as such events occur. The actual implementation of the analyzer is up to the runtime environment, enabling a broad range of use cases due to the respective runtime capabilities. In simulations, analyzers often rely on global knowledge to compute performance figures based on the whole system's state. The simulator version of the analyzer could, for example, consider the current overlay topology using global knowledge to examine the root cause of the empty buffer. During prototypical deployments, where global knowledge cannot be used, analyzers can annotate events with platform-specific data. In the streaming example, this could be information about the video provided by the media player, indicating short-lived increases in the bitrate due to a scene change in the video.

Listing 2: Retrieval and usage of custom analyzers.

```java
PlaybackAnalyzer a = Monitor.getAnalyzer(
    PlaybackAnalyzer.class);
a.onResume([...]); //Analyzer-specific method
```

By relying on platform-specific implementations of analyzers, each implementation can utilize the knowledge and tools of the respective platform at best. This enables evaluations with different scopes and goals to be conducted within the platform of choice through a single, unified instrumentation interface. The Simonstrator framework creates transparent proxies in case (i) the runtime provides multiple analyzers implementing the same interface, or (ii) the runtime provides no implementation of the analyzer. This enables simple analyzer retrieval and usage as shown in Listing 2.

In contrast to the application-specific analyzers, *metrics* are passive probes that provide access to system state. They implement a common interface as shown on the upper left-hand side in Figure 2, enabling retrieval of the current value as

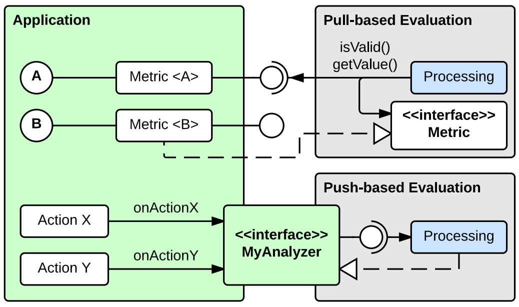


Figure 2: Concept of push- and pull-based instrumentation. Pull-based instrumentation uses the unified metric interface, whereas push-based instrumentation relies on application-specific analyzers.

well as a boolean indicating whether the probe's value can be considered valid. In the aforementioned example of a video streaming system, the current size of the playout buffer in terms of video chunks can be exported as a metric. The common interface, which is implemented by relevant metrics of a developed application (e.g., `Metric <A>` and `Metric <B>`), enables a set of automated processing and analyzing steps depending on the runtime environment. Utilizing a simulator's visualization capabilities to show an application's metrics can be easily achieved by bridging the metrics to the respective visualization tools of the simulator. During the development and debugging phases the concept of metrics provides vital insights into the system's current state. While metrics usually export numerical (or binary) states, they can be used to provide access to any kind of object, including application specific state containers. However, in this case, processing steps for the respective types need to be provided by the application developer, as the default set of processing steps are only defined for numerical values. Metrics and analyzers can be combined: this way, metrics can be processed upon specific actions triggered by an analyzer rather than solely based on periodic polling. This closely follows the concept of annotated data, but is more powerful when it comes to more complex analyzing steps involving the correlation of multiple probes. Following the design principles of the Simonstrator platform, one can bridge analyzers and metrics to the respective runtime's built-in functionality, for example metric visualization in simulators or logging capabilities on Android. This is further discussed in Section 2.2.

2.1.3 Hosts and Components

An individual device is modeled as a *host* within the Simonstrator framework. The host acts as container for any number of *components*. The composition of those components determines the host's functionality within the distributed system, comprising (i) the own developed components, (ii) already existing components from third parties, as well as (iii) components from the chosen runtime environment. Composition can be done during initialization or runtime, as components can request other registered components via the host.

One potential composition of a full streaming system is shown in Figure 3. In the example, an application developer defines the `Streaming` component interface and provides the respective implementation called System X, which is based on the defined interface. System X utilizes a generic publish/subscribe component as defined in the framework, with one implementation of the component being provided by the current runtime environment (System Y). Listing 3 shows the corresponding registration and binding calls to enable utilization of the publish/subscribe system within the streaming component. Both components utilize runtime environment components providing transport and network layer functionality. The specific realization of the pub/sub component furthermore utilizes a location sensor. Depending on the runtime environment, the location sensor component accesses real hardware (e.g., GPS on Android) or uses a movement model or data from a trace file.

Listing 3: Registration and retrieval of components.

```
// Pub/Sub registration (during initialization)
PubSub systemY = new SystemY();
host.registerComponent(systemY);

// Retrieval (in streaming component)
try {
  PubSub ps = host.getComponent(PubSub.class);
} catch (ComponentNotAvailableException e) {
  // [exception handling]
}
```

This enables natural development of complex distributed applications by composing and utilizing functionality of different components. Additionally, the approach enables isolated evaluation of single components by using stubs for other components. In the example shown in Figure 3, System Y can be replaced with any other component implementing the Pub/Sub interface. Such a component can be provided by the application developer or by the respective runtime environment. This enables transparent integration of third party libraries and platform-specific services such as Google Cloud Messaging on Android, as well as integration of global knowledge-based stubs in simulators. While this concept is already well-known in software engineering, it is largely neglected when it comes to the development and evaluation of scientific prototypes.

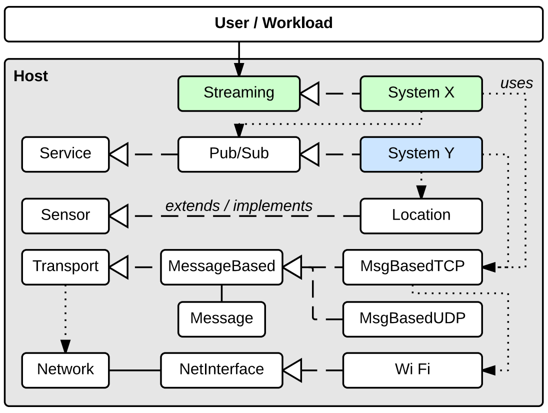


Figure 3: Host for a video streaming system resulting from the composition of platform and application components.

Following the concept of components and in contrast to [5], we do not provide a single dedicated networking abstraction. Instead, networking functionality is also realized via components. By providing a number of interfaces of different granularity within the framework, an application can, for example, first query for a MessageBasedTCP transport and relax the requirement to any kind of MessageBased transport in case of failure. This way, other network interfaces or transport protocols can be added easily to the framework and can then be utilized in the application code. Listing 4 illustrates binding and later usage of transport protocols. As first step, a network component is selected based on its name (or IP address), enabling support for multiple parallel network stacks. This is especially important to support applications for mobile devices, where multiple network interfaces are utilized (e.g., Bluetooth or Wi-Fi Direct for local communication and the cellular link for connections to a remote server). Protocol instances are created and listen for incoming messages on a given IP (and, thus, network interface) and port, invoking a message listener on each received message. Once the protocol is bound, it can be utilized as defined by the corresponding component interface. In the lower half of Listing 4, this is shown for the example of a MessageBased protocol. The component interface enables sending of messages in a *send-and-forget* manner by simply invoking send with the respective target address and a message object. For *request-reply* scenarios, it additionally provides the convenience functions sendAndWait and sendReply that mask the complexity of callback and timeout management for this common communication pattern.

Listing 4: Binding and usage of transport components.

```
// Initialization (binding)
net = host.getNetworkComponent().getByName(
    NetInterfaceName.WIFI);
MessageBased trans;
try {
  trans = host.getTransportComponent().getProtocol(
      UDP.class, net.getLocalInetAddress(), PORT);
  trans.setMessageListener(listener);
} catch (ProtocolNotAvailableException e) {
  // [exception handling]
}

// Usage (send and forget)
trans.send(Message msg, NetID rcvNet, int rcvPort);
// Usage (request-reply scenario support)
trans.sendAndWait(Message msg, NetID rcvNet,
    int rcvPort, ReplyCallback cb, long timeout);
```

All message-based transport interfaces operate on objects implementing the Message interface provided by the Simonstrator framework. The interface enforces the implementation of getSize, returning an estimation of the message size provided by the application developer. This method is used within simulation environments if actual payload is not required for the simulation model, thus avoiding serialization overhead and increasing scalability. In environments that need to consider *real* payloads, either for transmission or for calculation of the effective message size, the message object is serialized. This is further discussed in Section 3.1 for the use case of a live video streaming system. The Simonstrator framework relies on the Kryo library[2] for serialization. Using Kryo, an application developer *may* provide custom serializers using Java annotations. However, default handling within Kryo ensures out of the box serialization of classes with significant performance increases and reduced overhead when compared to the default Java serialization.

[2]Available under the BSD license at https://github.com/EsotericSoftware/kryo

Apart from interfaces for applications- and communication-related components, the Simonstrator framework contains interfaces for platform specific features such as access to sensors. Here, we largely mimic the API design of the Android platform, enabling developers to utilize a large fraction of the platform's features within their components via already familiar interfaces. One can easily connect sensors to traces or generators, such as the BonnMotion movement generator [2]. At the same time, real devices' location data can be used in prototypical deployments by redirecting calls to the respective Android API. This is further discussed in Section 2.2.3.

2.2 Runtime Environments

Instead of relying on one concrete simulator, systems developed on top of the Simonstrator framework can be deployed on a number of *runtime environments* that are part of the Simonstrator platform. A runtime environment can be a network simulator, an emulator, a testbed, or even a single mobile device. This enables evaluations of a common system from different perspectives and with different evaluation goals, relying on state of the art tools and platforms. In the following, we briefly discuss the runtime environments currently supported by the framework and provide pointers to the integration of additional runtimes. Case studies and sample evaluations showing the capabilities of the platform are discussed in Section 3.

2.2.1 *Simulation Runtime*

Event-based network or overlay simulators are a natural fit to our framework, as our abstraction of time and events follows the concept of event-based simulation. Whereas integration is straight forward for the Java-based PeerfactSim.KOM, an additional binding is required to communicate with the C++-based OMNeT framework, relying on a modified version of `JSimpleModule`[3]. Currently, the runtime environment for PeerfactSim.KOM features full access to all network and transport components (message-based) as well as integration with the movement models. Furthermore, the instrumentation capabilities of the Simonstrator framework are connected to the visualization and evaluation toolchain of PeerfactSim.KOM. The simulator already includes the notion of a *host* that manages several functional layers. The respective runtime extends the host to support dynamic binding and initialization of components. Custom components can be configured using PeerfactSim.KOM configuration files, as component creation is done in a factory pattern based on Java reflections.

As detailed in the previous section, the framework relies on a loose coupling of components. Therefore, when integrating the framework within a simulation environment, one does not need to provide implementations for all component interfaces. Instead, only those components that are relevant in the scope of the current simulation scenario need to be bridged to their respective simulation models. The models themselves as well as the scenario parameters are configured via the respective simulator. Consequently, the overall Simonstrator platform, comprising the framework with the currently utilized environment, does neither introduce nor require another configuration language. The only components that need to be provided by every runtime environment are the scheduling component responsible for handling time, events, and operations, as well as the *host* acting as a container for all components. The concept of simulation runtime environments enables the utilization of a wide range of models and platforms currently available to researchers through common interfaces.

[3]Available at `http://www.omnetpp.org/pmwiki/index.php?n=Main.JSimpleModule`

2.2.2 *Standalone Runtime*

As a foundation for testbed deployments and emulations, we provide the Standalone environment for the Simonstrator platform. Within this environment, the application runs as a simple Java program. By connecting the scheduling component to the operating system's clock, real-time operation of the system is achieved. The Standalone environment provides implementations for network components to support sending and receiving messages via TCP and UDP. In their default implementation, the network components rely on the KryoNet[4] library for network communication. A single-threaded deterministic scheduler is used, where all events (including incoming messages) pass through one queue before being executed. As container for all components, the Standalone runtime additionally provides an implementation of the `host` interface defined by the framework.

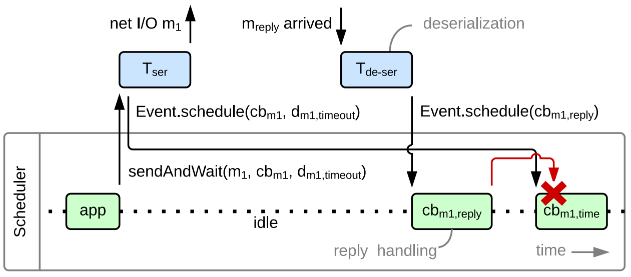


Figure 4: Decoupling of net I/O within the Standalone runtime in the *send-and-wait* scenario.

The runtime provides the scheduling component by relying on Java's `ScheduledExecutorService`, thereby ensuring that events are processed one after the other. As in event-based simulations, code has to be non-blocking. This property is achieved by utilizing callbacks and worker threads whenever blocking functions (such as net I/O) are called in the runtime environment. For net I/O, this is already taken care of by the KryoNet library. In order to avoid concurrent execution of application code, incoming messages are added as new immediate events into the scheduling queue. As soon as the event is executed by the scheduler, the respective event handler (usually a message handler in the application code) is notified and can process the message.

Figure 4 illustrates the behavior of the scheduler for the *request-reply* scenario outlined in Section 2.1.3. The application code invokes `sendAndWait` on the transport component, passing a message m_1 to send, a callback cb_{m1} as well as a timeout. The runtime implementation of the transport component stores the callback object cb_{m1} and spawns a

[4]Available under the BSD license at `https://github.com/EsotericSoftware/kryonet`

new thread $\mathrm{T}_{\mathrm{ser}}$. The `sendAndWait` method returns immediately afterwards, thereby ending the execution of the event in the scheduler and allowing the next event to be processed. The thread $\mathrm{T}_{\mathrm{ser}}$ takes care of scheduling the timeout event $\mathrm{cb}_{\mathrm{m1,time}}$ as well as serializing the message and passing it on to KryoNet. In the example, a reply m_{reply} arrives before the timeout fires, resulting in a new thread $\mathrm{T}_{\mathrm{de\text{-}ser}}$ that takes care of deserialization. The message object is then enqueued for further processing by scheduling an immediate event. In our example, the scheduler is currently idle, leading to the immediate execution of the corresponding event. As part of the callback handling provided by the `MessageBased` transport component, the corresponding timeout event $\mathrm{cb}_{\mathrm{m1,time}}$ is disabled and will not be processed further.

2.2.3 Android Runtime

Extending the Standalone environment, the Android environment adds an application stub and bridges Android-specific interfaces, such as access to sensors. It serves as the basis for the development of user interfaces and instrumented applications. As the Android runtime relies on the Standalone runtime, both environments are fully compatible w.r.t. message serialization and net I/O. This enables heterogeneous deployments with both, mobile Android clients and fixed computers running on a testbed such as PlanetLab[5] or G-Lab/ToMaTo[6].

The Android runtime serves two purposes, depending on the scope of the intended evaluation. For technical evaluations, a fully fledged Android application is usually not required. In such scenarios, the runtime supports researchers and developers by providing basic information about instrumented components through *metrics*. For user studies, an easy accessible user interface is required, best developed using standard Android development tools. By connecting interactions with the user interface to the respective calls of the components' APIs, and by utilizing *analyzers* to pass relevant component state to the user interface, a clear separation of the user interface and the functional components is achieved. In addition to the simplification of the application development, it also ensures that up-to-date component implementations can directly be used within prototypes. Furthermore, applications developed with the framework can also run on the default Android emulator. This provides easy ways to control specific parts of the Android application, such as injecting location data though our sensor interfaces.

Using the instrumentation features of the Simonstrator framework, one can easily record sensor data or user interaction patterns obtained during a field study. Such data can then be fed back to simulation models for larger-scale evaluations. Possible applications include the recording of individual users' movement traces (c.f. Section 3) as well as gathering statistics about the utilized network connection (e.g., signal strength) to later correlate that data with user behavior patterns.

2.2.4 Other Runtime Environments

Besides adding other network or overlay simulators by bridging the respective components to the simulator's models, one can easily extend the Standalone runtime to utilize generators for specific data. One example is the BonnMotion [2] movement generator, which can be used to generate node mobility traces. Instead of relying on real GPS data, one can feed trace values generated by BonnMotion into the application by providing the respective implementation of the location sensor component. This is especially valuable during prototypical deployments relying on automated sequences of sensor data (in this case movement) to trigger specific application behavior. By relying on the location sensor abstraction provided by the framework, the underlying implementation is completely transparent to the application or other components utilizing the sensor.

[5]`https://www.planet-lab.org/`

[6]`http://master.tomato-lab.org`

3. EVALUATION

Within this section, exemplary use cases of the proposed Simonstrator platform are discussed. We especially want to highlight the benefits of the component-based approach for the development and evaluation of complex distributed systems on heterogeneous platforms. Therefore, two use cases are analyzed in the following: (i) a distributed live video streaming system, and (ii) a publish/subscribe middleware for augmented reality games. For each use case, the utilization and configuration of respective components from the Simonstrator platform are discussed together with the lessons learned. As the Simonstrator platform utilizes existing simulators and their models through the concept of runtime environments, we do not intend to compare the performance of different simulation models.

3.1 Live Video Streaming

Video streaming is a prominent application on today's mobile devices. To deal with an increasing user demand and simultaneously compensate for sudden increases in viewer numbers, CDN providers such as Akamai build large overlay-based distribution structures. They span both hundred thousands of managed servers [14] as well as actively contributing clients at the edge of the network, i.e. applying peer-to-peer (p2p) technology for an efficient distribution of content [21]. As an example for the latter case, consumers of a video stream can contribute some of their upload bandwidth to support the overall delivery process as well as reduce the load on the content provider and the CDN infrastructure. While there have been countless proposals for distributed streaming systems in fixed networks, efficiently supporting typical live streaming use cases remains a challenging problem. Furthermore, the incorporation of mobile devices in p2p systems poses an additional challenge.

If a mobile device is connected via a cellular link (i.e. 3G/4G), the upload bandwidth is not only limited but also expensive in terms of energy consumption on the device. Therefore, uploading video data via the cellular link is usually not a viable option in typical mobile environments. However, in group scenarios, the video data can be shared with other nearby devices, e.g. using Wi-Fi Direct. If done in an intelligent manner, the overall bandwidth and energy consumption of a group of devices can be reduced. However, tablets tend to be used a lot at home, typically connected to the home network using Wi-Fi and often being connected to a charger. Suddenly, even mobile devices can become fully-fledged peers in a distributed system, contributing *cheap* upload bandwidth and strengthening the overall streaming system. A more severe challenge for distributed streaming approaches is the ability of a system to cope with sudden increases in the overall number of viewers, so called *flash crowds*. In this context, scalability of the overall system, i.e. the ability of the streaming system to *quickly* leverage upload bandwidth of new peers to address the fierce competition for available resources is critical and an important focus of research in this area.

These different challenges open a wide area of different evaluation scenarios that are to be considered when designing and studying new streaming systems. One recently proposed live streaming system that aims at supporting heterogeneous clients and to handle flash crowd events is Transit [20, 13]. It was evaluated using the Simonstrator framework and, this way, successfully evaluated using large-scale simulations and has shown to work also as prototype running on mobile devices for small-scale demonstrations. Currently, mid-scale testbed evaluations are in preparation to complete the picture.

Figure 5 illustrates one host within the Transit overlay and its usage of framework components when transmitting video blocks. These components are then mapped to the respective runtime environment, here shown for the simulator PeerfactSim.KOM as well as the prototypical deployment on Android and on testbed PCs. Note, how real video data as well as packet serialization is only used in the prototypical deployments. Within simulations, the `Message` interface is used to obtain the size of a message in bytes without actually creating the corresponding payload (cf. Sec. 2.1.3). This example illustrates the choice of relevant effects within the considered scenario: the actual video payload is assumed to have no influence on the macroscopic system behavior during a flash crowd and therefore is omitted when studying streaming performance and structural properties in large-scale scenarios.

Figure 6 shows exemplary simulation results for synthetic flash crowd scenarios. The number of active peers as well as the arrival rate during two flash crowds with a peak of 2,500 concurrent peers and different intensities of arrival rates is shown in Figure 6a. In Figure 6b the performance of two configurations under the FC2-workload is shown in terms of video stalling events, i.e., interruptions in playback. The topology optimizations in the configuration TopT lead to a significant decrease of such interruptions during the flash crowd by lowering the average delivery tree depth (c.f. Figure 6c) and, thus, reducing the impact of peer arrivals and departures. The objective of these studies was to investigate the scaling characteristics of an improved version of the Transit system in extreme scenarios and evaluate different system variants and configurations. For this purpose, simulative evaluations were chosen (i.e. a macroscopic view) as only feasible approach to investigate a large number of different configurations with well-defined and controlled large-scale streaming scenarios.

Relying on the Simonstrator framework and, thus the same core implementation, currently, mid-scale testbed experiments (i.e. with up to several hundred headless clients) are in preparation to complement the simulation results described above. The objective is still to study the scalability behavior of the system but under less resource-restricted conditions, now also considering more practical factors, such as variable-sized video chunks. For these testbed experiments, a recently extended version of the German-wide G-Lab testbed is used. The same version of the streaming software could be also run on PlanetLab or similar testbeds with no changes. For these experiments, the streaming system is run on Linux-based, headless testbed machines. While the data transmission of video packets is done in a full manner, the video playback at the clients is simulated. A virtual video player probes the buffer of the local streaming client for the availability of the video data and triggers stalling events on unavailable data. This allows to assess the streaming system performance across all clients in the same way as for the simulation studies, relying on a common analyzer interface as introduced in Section 2.1.2.

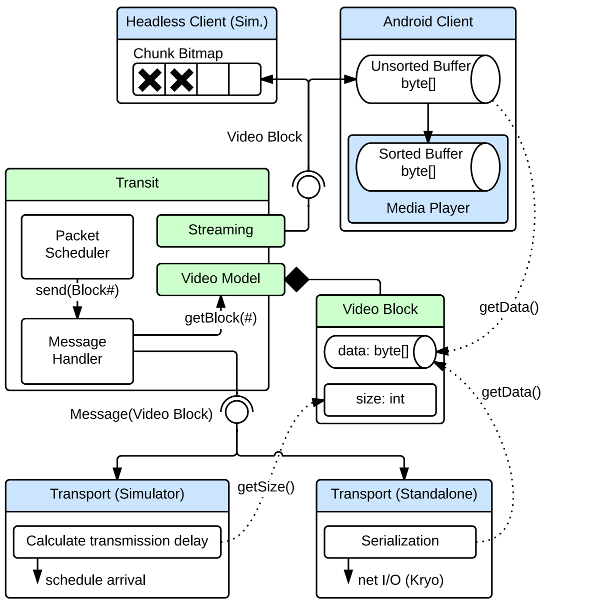
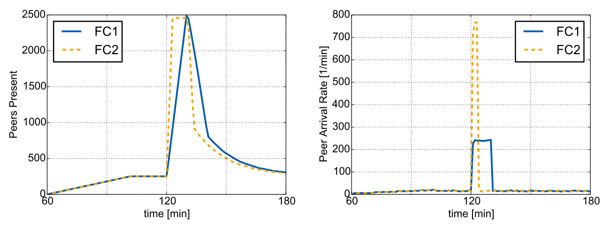


Figure 5: Handling of video data within Transit. On Android, real video payload is transmitted and passed to the media player, whereas simulations rely on `getSize` provided by the `Message` interface to increase scalability.

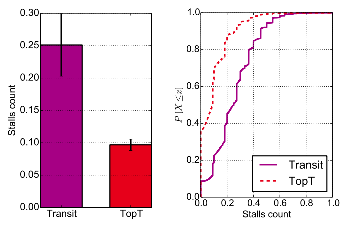
A step further was taken by bundling the streaming software in an Android project, where, again thanks to the Simonstrator framework, it is used as library with no need to change the actual implementation. Here, the virtual video player was replaced by a real video player library as part of an Android application. This app is used for demonstration purposes [12] and was successfully shown at different occasions to demonstrate the adaptive behavior of the streaming system. For the demonstration, the focus is on small-scale scenarios and the user-centric view on the system. Here, the graphical user interface and the interactive behavior of the system plays a more important role than other aspects studied in the simulation or headless testbed evaluations.

3.2 Augmented Reality Gaming

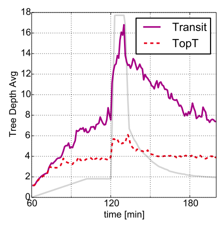
Following the success of Google's augmented reality (AR) game *Ingress*, location-aware games and applications are be-



(a) Simulated flash crowd scenarios of different strengths.

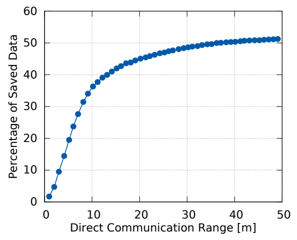


(b) Performance comparison based on video stalling events.

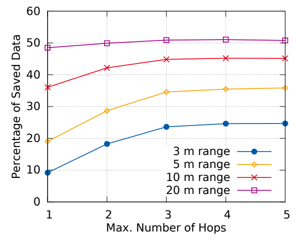


(c) Influence on topology characteristics.

Figure 6: Typical scenario and results for streaming simulations comparing different streaming system variants for the FC2 workload (adapted from [13]).



(a) Transmission ranges.



(b) Multi-hop distribution.

Figure 7: Potential for direct local communication assuming different communication ranges as well as multi-hop distribution as obtained from the field study.

coming increasingly popular. One important property, especially of AR games, is the spatial relevance of information: interactions with the game mostly affect other players that are in physical proximity. Today, these communication characteristics are not reflected by the underlying communication system. Instead, data is sent to a centralized infrastructure (e.g., cloud data centers), although it is of local relevance. This leads to increased latencies, effectively prohibiting highly interactive applications. In previous work [11], we propose and evaluate a system for locality-aware messaging that utilizes local communication between nearby devices to augment communication via the central server.

In contrast to the video streaming example, the AR scenario is targeted solely towards mobile devices, relying on sensor data for location updates as well as on local communication, using Wi-Fi and Bluetooth. Furthermore, user movement and behavior is of primary interest as it directly determines the workload and, thus, the performance of the system. To collect realistic user traces that can then be used for larger scale simulations, an actual game is implemented using the Simonstrator Android runtime [10]. The game communicates with a central server (running within the standalone environment) using our communication middleware. The middleware is instrumented to log all events that are triggered by user interaction with the game. These measurements are stored locally on each device for later processing. Evaluating data gathered during these field studies serves two main purposes: (i) improving the workload model in our simulations to enable assessment of different local communication strategies and (ii) later evaluate the strategies under realistic conditions, especially w.r.t. the wireless communication medium.

To study user behavior and the potential savings obtained through local communication, we conducted a field study with 40 players playing the game in a public park and its direct surroundings. The players were divided into two groups that each tried to win the session by conquering locations, similar to portals in Google's Ingress. The locations were preselected based on map data and included public sights and monuments spread all over the park. In order to conquer a location, a player spawns units that follow his real-life movements and attack towers built by players of the enemy group. Thus, the chance of winning increases if more players agree on a common target to attack. However, groups can split up to speed up the process by conquering multiple (potentially weaker) targets in parallel. Two findings affecting the design of the communication middleware are shown in Figure 7. Based on the movement data and the recorded communication events of two sessions of one hour each, (a) shows the percentage of events that could be distributed locally over one hop, given a specific communication range. In (b), the corresponding results for multi-hop communication are shown for selected communication ranges, indicating that for the observed user behavior the efficiency of multi-hop communication saturates at roughly three hops, independent of the communication range. Even with single-hop communication and a range of 3 m, 10 % of the overall data can be distributed locally, motivating low-power local transmission of events via Bluetooth.

The observed effects and characteristics (cf. Figure 7a and 7b) motivated us to investigate the influence of direct communication for the transmission of events to nearby players. As published in [11], we examined the impact of different local communication protocols, which rely on Wi-Fi ad hoc, on our communication middleware by means of simulation. In a prospective step, we plan to analyze the obtained movement data from the field study to improve the utilized mobility model for a more accurate reflection of user behavior in the AR game.

Adding it up, the example shows the benefits of combined simulative and prototypical evaluation, especially when there are many environmental factors potentially influencing a system's behavior and performance. To ease such combined evaluations of mobile applications, the Simonstrator platform provides abstractions for device-specific features, such as access to sensors or multiple communication interfaces. This way, the Android prototype is kept up to date with the respective simulation model without additional implementation efforts.

4. RELATED WORK

Related to the Simonstrator platform comparable simulation platforms have been developed that enable combined simulative and prototypical evaluations. In the following, we start with an overview on platforms that target specific types of networks, comprising delay tolerant networks (DTNs), p2p networks, and wireless sensor networks (WSNs). Subsequently, we present related work on solutions that combine virtual or emulated machines with simulators and complete this section with a review of common network simulators.

ONE [8] is a simulator for DTNs, focusing on node movement and the resulting contact characteristics to enable evaluation of DTN protocols. The ONE simulator exports the DTN2[7] interface, enabling the integration of real-world applications into the simulation environment. Additionally, the simulator can communicate with DTN2 bundle routers, thereby enabling utilization of its connectivity models in testbed deployments. Our concept of runtime environments further extends the potential deployment options when compared to the ONE simulator. Furthermore, the presented Simonstrator framework with its component-based composition of experiments is not limited to one specific application scenario (DTNs). Instead, our framework enables a broader scope of evaluations with a common research prototype.

ProtoPeer [5] is closely related to our proposed framework, as Galuba et al. introduce a lightweight programming interface for developed applications on top of a simulator or a prototype. The proposed interface is limited to scheduling and network-related functionality and enforces a strict horizontal split between application and platform. The concept of Peerlets allows the composition of applications out of several functional building blocks, similar to our proposed components. However, Peerlets in ProtoPeer are limited to the application layer and cannot be provided by the platform itself. This limits the usage of platform-specific features, such as utilizing location data provided through the Android API. Therefore, the ProtoPeer approach is sufficient for fixed p2p networks that exclusively rely on the interconnection of and communication between fixed hosts, as illustrated with the Chord example by the authors. In contrast, it does not enable design and evaluation of today's mobile applications, as the abstraction in ProtoPeer is limited to only time and network-related features.

WSN research also focuses on both, simulative and prototypical evaluation of small-scale and large-scale behavior. Therefore, a number of WSN testbeds have emerged, that enable protocol assessment within a defined setup and at specific scale. For early insights as well as scenarios where not suitable testbed is available, researchers rely on simulation tools that often come bundled together with the sensor nodes' operating system. One prominent example is TOSSIM [9], providing a complete simulation environment for applications built on the TinyOS operating system. Applications and protocols written for TinyOS can be deployed on real hardware or being simulated within TOSSIM without changes to the application code itself. The compile process ensures that instead of being executed on real hardware, calls are directed to the respective simulation models. COOJA/MSPSim [4] goes one step further by enabling simulation and emulation of WSNs with heterogeneous operating systems. The authors argue that testing of interoperability is a key requirement when developing applications and protocols for heterogeneous sensor networks. This claim can also be applied to today's applications, running on a broad range of heterogeneous devices, ranging from mobile to fixed devices. By enabling component-based composition and evaluation of a system on different runtime environments, the Simonstrator platform helps to consider heterogeneity within the experiment setup.

Werthmann et al. [19] propose VMSimInt, utilizing virtual machines (VMs) integrated into a simulation framework. By controlling the VM's time and clock speed, as well as access to network socket operations, they enable time-scaled experiments while providing full access to the operating system's features. The focus of VMSimInt is on transport and network layer protocols, where the VM-based abstraction benefits from the realistic behavior of the network stack implemented in the operating system. Regarding the reproducibility of experiments, the authors do not control the random number generator of the host system. They argue that their approach still enables deterministic evaluation of TCP mechanisms, as protocol behavior is only affected by the system's clock in this case. However, VMSimInt adds a considerable amount of overhead. Consequently, it can not be used to evaluate larger distributed systems such as the streaming system discussed in Section 3. Furthermore, it does not provide an abstraction for heterogeneous platforms, as the studied application has to be programmed for the respective OS. Still, one could integrate applications developed with the Simonstrator framework into VMSimInt to study network-layer effects by using the Standalone runtime environment inside a virtual machine.

SliceTime [18] constitutes an emulation platform that combines event-based simulations with virtual machines to examine macroscopic as well as microscopic effects on the evaluated communication protocol. The platform consists of three components, comprising a synchronizer, a simulator, and one (or multiple) virtual machines. The synchronizer is used to coordinate and synchronize the execution of the considered communication protocol that is executed on the simulator and the virtual machines. Dependent on the speed of the simulation, the synchronizer either throttles the simulator if simulations run faster than real time or stops the virtual machines otherwise. SliceTime relies on the network simulator ns-3 [6], which facilitates to reuse the code from simulations for the deployment on the virtual machines. As is the case with VMSimInt, SliceTime does not provide an abstraction for heterogeneous platforms, especially considering mobile devices.

By providing the respective runtime environments for the Simonstrator framework, one can integrate other simulation engines and utilize existing models during evaluation. In this work, we discussed the integration of the PeerfactSim.KOM overlay simulator and the OMNeT++/INET network simulator. As only the core framework functionality (e.g., scheduling and the host implementation) has to be provided by the runtime implementation, other simulators can be integrated easily.

[7] `http://www.dtnrg.org/wiki/Code`

5. CONCLUSIONS

Distributed mobile applications and the underlying communication systems are subject to a high number of environmental factors influencing their performance. To assess the impact of these factors at different scales, different types of simulation as well as prototypical deployments are necessary. In this paper, we proposed the Simonstrator platform, consisting of the component-based Simonstrator framework as well as runtime environments for existing simulators, testbeds, and the Android mobile platform. The Simonstrator platform especially supports design and evaluation of applications for heterogeneous mobile devices by enabling access to device-specific features such as sensors or multiple network interfaces (e.g., Bluetooth and Wi-Fi Direct). The component-based design in conjunction with different runtimes and powerful instrumentation capabilities enables evaluations of complex distributed applications under a wide variety of evaluation scopes. Systems developed with the proposed framework can run on any of the provided runtime environments without modifications to their code.

We demonstrate the capabilities of our platform by discussing live video streaming and mobile augmented reality gaming as two exemplary use cases. Both use cases highlight the benefits of combined evaluations at different scales. Early user studies based on prototypes motivate design decisions for further research, while fast feedback of simulation models into the prototype helps in understanding the system's behavior under realistic settings.

The Simonstrator platform is being used in a number of projects and benefits from their contributions. Currently, the Android runtime and the framework is being extended to support additional Android-specific features such as Near Field Communication (NFC). Furthermore, deeper integration with automated deployment tools such as Plush [1] is being examined, allowing easier setup of testbed experiments and collection of evaluation results. Finally, the OMNeT runtime environment is being finished for release, which includes integrating OMNeT's visualization framework.

6. ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation (DFG) as part of projects B01/C02/C03 within the Collaborative Research Centre (CRC) 1053 – MAKI. Ralf Steinmetz has been partially supported by a Chair of Excellence from University Carlos III, Madrid, Spain.

7. REFERENCES

[1] J. R. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote control: Distributed application configuration, management, and visualization with plush. In *Proc. LISA*, 2007.

[2] N. Aschenbruck, R. Ernst, E. Gerhards-Padilla, and M. Schwamborn. Bonnmotion: a mobility scenario generation and analysis tool. In *Proc. SIMUTools*, 2010.

[3] I. Baumgart, B. Heep, and S. Krause. Oversim: A scalable and flexible overlay framework for simulation and real network applications. In *Proc. IEEE P2P*, 2009.

[4] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón. Cooja/mspsim: interoperability testing for wireless sensor networks. In *Proc. SIMUTools*, 2009.

[5] W. Galuba, K. Aberer, Z. Despotovic, and W. Kellerer. Protopeer: A p2p toolkit bridging the gap between simulation and live deployement. In *Proc. SIMUTools*, 2009.

[6] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 project goals. In *Proc. WNS2*, 2006.

[7] R. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 2008.

[8] A. Keränen, J. Ott, and T. Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In *Proc. SIMUTools*, 2009.

[9] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proc. ACM SenSys*, 2003.

[10] B. Richerzhagen, M. Schiller, M. Lehn, D. Lapiner, and R. Steinmetz. Transition-enabled event dissemination for pervasive mobile multiplayer games. In *Proc. WoWMoM*. IEEE, 2015.

[11] B. Richerzhagen, D. Stingl, R. Hans, C. Gross, and R. Steinmetz. Bypassing the cloud: Peer-assisted event dissemination for augmented reality games. In *Proc. IEEE P2P*, 2014.

[12] B. Richerzhagen, S. Wilk, J. Rückert, D. Stohr, and W. Effelsberg. Transitions in live video streaming services. In *Proc. ACM VideoNEXT*, 2014.

[13] J. Rückert, B. Richerzhagen, E. Lidanski, R. Steinmetz, and D. Hausheer. Topt: Supporting flash crowd events in hybrid overlay-based live streaming. In *Proc. IFIP Networking*, 2015.

[14] R. K. Sitaraman, M. Kasbekar, W. Lichtenstein, and M. Jain. Overlay Networks: An Akamai Perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons, 2014.

[15] A. Stetsko, T. Smolka, V. Matyas, and F. Jurnečka. On the credibility of wireless sensor network simulations: evaluation of intrusion detection system. In *Proc. SIMUTools*, 2012.

[16] D. Stingl, C. Gross, J. Rückert, L. Nobach, A. Kovacevic, and R. Steinmetz. PeerfactSim.KOM: A simulation framework for peer-to-peer systems. In *Proc. HPCS*, 2011.

[17] A. Varga et al. The omnet++ discrete event simulation system. In *Proc. ESM*, 2001.

[18] E. Weingärtner, F. Schmidt, H. V. Lehn, T. Heer, and K. Wehrle. Slicetime: A platform for scalable and accurate network emulation. In *Proc. NSDI*, 2011.

[19] T. Werthmann, M. Kaschub, M. Kühlewind, S. Scholz, and D. Wagner. Vmsimint: a network simulation tool supporting integration of arbitrary kernels and applications. In *Proc. SIMUTools*, 2014.

[20] M. Wichtlhuber, B. Richerzhagen, J. Rückert, and D. Hausheer. Transit: Supporting transitions in peer-to-peer live video streaming. In *Proc. IFIP Networking*, 2014.

[21] M. Zhao, A. Chen, Y. Lin, and A. Haeberlen. Peer-Assisted Content Distribution in Akamai NetSession. In *ACM IMC*, 2013.