Scalable Context-Aware Development Infrastructure for Interactive Systems in Smart Environments

Tobias Eichler
Department of Computer Science
Hamburg University of Applied Sciences
Hamburg, Germany
Tobias.Eichler@haw-hamburg.de

Susanne Draheim
Department of Computer Science
Hamburg University of Applied Sciences
Hamburg, Germany
Susanne.Draheim@haw-hamburg.de

Christos Grecos
Department of Computer Science
Central Washington University
Ellensburg, WA, USA
Christos.Graikos@cwu.edu

Qi Wang
School of Engineering & Computing
University of the West of Scotland
Paisley, UK
Qi.Wang@uws.ac.uk

Kai von Luck
Department of Computer Science
Hamburg University of Applied Sciences
Hamburg, Germany
Kai.vonLuck@haw-hamburg.de

***Abstract*—Context-aware systems for smart environments can be very complex and demanding for developers especially in distributed computing and communication environments. We propose a new development infrastructure, that targets this challenge by improving the general system's scalability and traceability. The infrastructure has been developed for and tested in two research labs for smart environments and human computer interaction. First measurements show that the platform has high scalability and low message latency that is perfectly suitable for interactive projects and virtual reality experiments.**

***Index Terms*—context awareness, scalability, middleware, multi-agent systems, smart environments**

I. Introduction

Systems for smart environments are often based on multi-agent systems, which are highly parallel, distributed and message based. The complexity in these systems is usually not caused by individual software components, but by the combination of a large number of communicating agents [1]. To handle this complexity a middleware and special development tools are investigated in this PhD research. We present an agent-based, context-aware development infrastructure that is particularly designed for smart environments and fast development cycles. The messaging is based on the publish/subscribe model, which is suitable for distributed middleware [2], to handle fast event propagation and to support easy access for developers to all messages. The proposed system is employed as a base for multiple research projects in our laboratories. On top of this infrastructure we describe first approaches to improving the development process of such systems. Sections II and III describe the targeted environment and the requirements for the system in more detail. Section IV contains an overview of the overall system architecture, and Section V reports latency and scalability evaluation results. In Section VI we present one of our case studies we conducted based on the developed infrastructure. Finally we highlight the plan for future work in Section VII.

II. Environment

Our approach is based on the analysis of projects in two research laboratories. The main topics of our research group are smart environments, human computer interaction and virtual reality (VR).

The Living Place Hamburg [3] is a 144 m^2 fully functional loft style apartment on the campus of the Hamburg University of Applied Sciences (HAW). The Living Place is separated into the living area, where experiments with smart objects and pervasive systems are conducted, and a control room for evaluation purposes. The Creative Space for Technical Innovations (CSTI) is a laboratory for experiments with innovative technologies for smart environments, smart objects and new interaction concepts at the HAW. The CSTI approach is based on supporting diverse experiments with proof-of-concept prototypes developed in short cycles [4].

III. Requirements

There are a number of requirements resulting from the research topics and the targeted environment. One of the main goals of the system is to enable developers to change components and add new ones easily. This is needed for rapid prototyping of context-aware software components. In addition to the basic requirements for a middleware identified in [5], like support for heterogeneity and tolerance for component failures, there are a number of requirements that are new or different based on our specific use case.

- **Traceability and Control** - The system is used in a research context and should be easily expandable. All messages between agents have to be accessible by all permitted components and developers due to subscriptions. The messages have to be human readable or can be transformed into human readable information. At any time it has to be possible to send messages from outside of the system to all existing groups for

978-1-5386-3839-2/17/$31.00 ©2017 IEEE

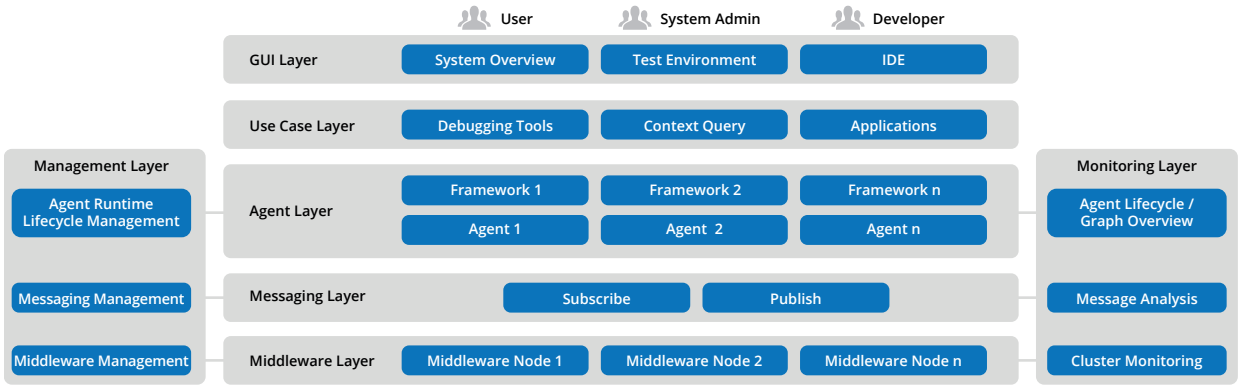

Fig. 1: System architecture overview

debugging purposes. Agent information, group names and their subscriptions have to be available at runtime to support system monitoring and inspection

- **Support for Mobility** - The system is able to automatically spawn new agents in a controlled environment. This can be used to react to context changes or as a mechanism for fault tolerance. Additionally this feature is used to create controllable testing environments.
- **Agent Management** - The system has to provide runtime information like component names, identifiers, the execution environment and software dependencies, about all agents. The life-cycle of agents has to be manageable by the system. An agent can be migrated or cloned to other runtime environments over the network.
- **Scalability and Latency** - The system has to handle components with frequent user interaction and high message throughput to support sensors for gesture recognition and head-mounted displays. Gesture detection sensors can produce a huge amount of data that needs to be handled by the system.

IV. The Proposed Architecture

The overall architecture of the system can be partitioned into five layers, as shown in Fig. 1. The bottom layer, called Middleware Layer, is the foundation for the messaging of the system. It delivers messages over TCP, SCTP and Websocket connections, and manages and monitors the system. The next layer of abstraction is the Messaging Layer, which implements the publish/subscribe based group communication. Messages can be sent to arbitrary groups, which can be subscribed by all agents in the system. Agents use the Messaging Layer to offer different services to the rest of the system. Multiple agents can work together to implement more complex tasks, e.g. the aggregation and interpretation of sensor data. This agent groups are located in the Agent Layer. The Agent Layer supports the Use Case Layer, which contains all applications, the context handling and the debugging tools. The interaction of users, system admins and developers with the system is handled by the GUI Layer. The platform consists of all components in the system, including the middleware, runtime environments and all agents.

A diagram of the platform architecture can be seen in Fig. 2. The platform consists of at least one middleware node. All nodes manage their cluster membership by heartbeat messages. They provide a publish/subscribe based communication service for the agents. Additional services such as monitoring or failure handling are implemented as agents. The platform can be controlled by a web interface, which is a fully featured agent implemented with web technologies. External messaging systems, like Java Messaging Services (JMS) or other middleware systems can be transparently connected to the system.

A. Interface Libraries and Framework

Interface libraries are used to specify the possible messages between agents. This allows developers to use messages in integrated development environments like a common method call including auto-completion and type checking. The libraries are managed by an artefact managing system and are versioned to avoid conflicts. The middleware can check at runtime if a subscription uses the correct interface with the correct version.

An interface library generator is used to generate libraries in multiple target languages based on a simple message DSL. These libraries include the native message representations (e.g. Java classes and interfaces) and the serialization code for all supported methods. Currently supported languages are Java, Scala, C++ for native code and Javascript, which allows the easy development of graphical web interfaces for services. For the development of new agents a framework is provided, which handles the connection to a middleware node, implements an auto-reconnect mechanism and hides the serialization of messages.

All agents communicate exclusively with messages over the group based publish/subscribe service provided by the middleware. Direct communication (Unicast) can be achieved by sending a message to a group with a single subscriber. This ensures that all communication channels can be monitored at any time.

Groups exist if there is at least one subscriber. Messages sent to non existing groups are discarded. The messaging follows a

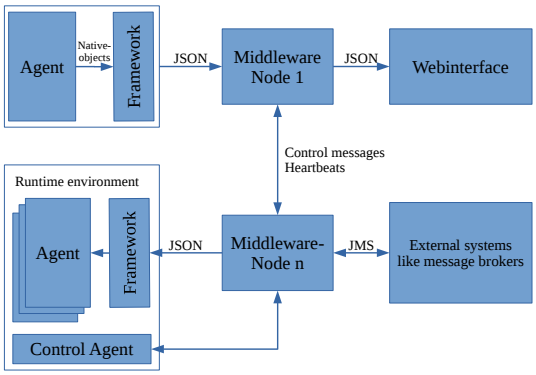

Fig. 2: Platform Architecture

at-most-once delivery semantic. If other semantics are needed they can be implemented based on the provided services. A redelivery or reordering of messages could slow down the system and increase message latency. Many components, e.g. many sensors that can have a high message frequency, do not need a reliable message delivery.

The system supports multiple message formats. The default is JSON. This format is especially useful during testing, because it can be easily read by humans and is supported by many programming languages. In production systems a binary json format is used, which results in much smaller messages, that can be generated and parsed a lot quicker. If an even smaller format is needed other formats like Google Protobuf can be used. The configuration can be generated out of the interface definition or defined manually. If this message formats are used, components can only deserialize messages with access to the message definition.

The used message format can be set by configuration or automatically by the system. All components can communicate which formats they understand. The middleware can translate message formats if needed to allow communication between agents with different capabilities.

B. Runtime and Event Sourcing

For the dynamic deployment and controlled execution of the software components, a runtime environment is needed. It can take software artefacts and execute them with provided configuration on a specific host. This allows the system to create and stop agents dynamically at any time. The running components can be monitored by the environment allowing notifications on failures or shutdowns.

Agents can be moved between runtime environments if they are stateless or have a persisted execution state. Each message sent to an agent is saved to an distributed database. This Event Sourcing can be enabled by the developer of an agent. If the runtime environment fails, another node can restart the agent, replay all messages and continue from there. To optimize this procedure each agent can save snapshots of its state, so that on restart of an agent the node only has to load the latest snapshot and newer messages. All messages that are older than the latest snapshot are deleted from the database. If the state of an agent is persisted with this feature, the current state of this agent can be analyzed by the developers and all permitted system components.

V. Latency and Scalability Evaluation

For our user interaction projects, a low message latency is very important. This platform is being used for virtual and augmented reality experiments, where the latency between the tracking sensors and the visualization is critical [6]. For example head-mounted displays with latency issues can cause severe motion sickness.

The testing environment for the measurements consists of eight computers connected over an local 1Gbit network switch. One of the hosts is used as a control node with the database for the test system. This host is used to deploy any desired number of agents on the other hosts. There are two different agents that are used for this latency test. One sends a message every configured time period and the other receives the message and sends it back. This means that the full round trip time including the serialization an deserialization is measured. Each agent pair uses an separated group for the communication.

Fig. 3 shows the message latency on eight middleware nodes and seven runtime environments with a variable number of agents sending a message every four to five seconds. This means there are up to 27,000 messages per second passed between the agents without counting the system messages like the heartbeats between the middleware nodes.

The average message latency with 120,000 agents is under 12 milliseconds and reaches a maximum of 30 milliseconds. This is an extremely large number of agents for a smart environment with such a high message frequency. In comparison, in most of our targeted environments, there are fewer than 1,000 concurrent agents running at a time.

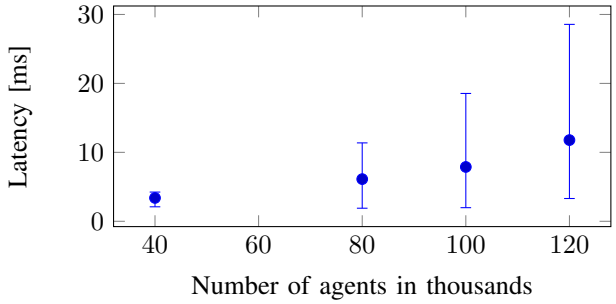

Fig. 3: Test with different numbers of agents, eight middleware nodes and seven runtime environment.

Smart environments can consist of many communicating agents with different message traffic. The platform have to scale with the size of the system. It is possible to add a new middleware node at any time to decrease the message latency of the system if it is overloaded. Fig. 4 shows the result of a test with a fixed number of 10,000 agents on a variable number of middleware nodes with runtime environments. Each agent sends a message every 300 to 400 milliseconds.

It can be seen that two middleware nodes can handle the 10,000 agents with about 60 ms message latency. However,

there are messages that require twice as much time, because the system is running on maximum load. With four and six middleware nodes the latency and the deviation decreases significantly to about 5.5 milliseconds, which is suitable for our purposes.

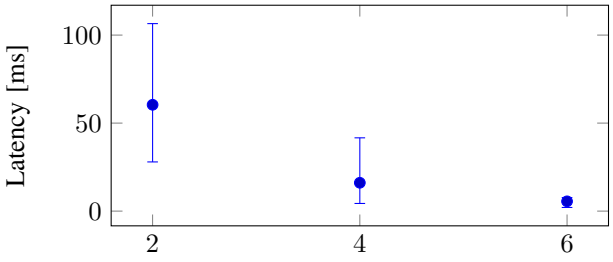

Fig. 4: Test with 10,000 agents and different numbers of middleware nodes with runtime environments.

VI. Case Study: Sensor Fusion for VR Interaction

The proposed platform was tested with multiple projects in the presented laboratories. One of many case studies, that combines most of our research topics is the sensor fusion project for omnidirectional skeleton tracking.

Skeleton tracking of body parts can be useful to implement gesture recognition. Popular sensors like the Microsoft Kinect 2 have an interaction area of only a few meters. For the implementation of interaction areas in the size of a full apartment, like the Living Place Hamburg, or the implementation of omnidirectional tracking multiple sensors have to be employed.

One use case for this sensor fusion is walking in place detection for natural interactions with VR worlds [7]. To implement this walking in place detection it was necessary to measure the position of the legs and feet of one person with low latency. The tracking had to be omnidirectional because the user can freely rotate herself. To achieve this, we had to use multiple Kinect 2 sensors for the skeleton detection. It was feasible to implement the needed components for this projects in a few weeks based on our messaging platform.

We are currently working on an generic tracking system, which can handle multiple static and dynamic coordinate systems with automatic calibration. One example for a dynamic coordinate system is a hand motion sensor, which is mounted on a head-mounted display. The display is tracked in one coordinate system together with controllers etc. and the mounted sensor has a variable origin based on that. This is needed to implement multiple user interaction with VR environments. The platform handles the coordination between the distributed components and helps the developers to debug this system.

VII. Conclusion and Future Work

We presented an agent based development infrastructure for smart environment with publish/subscribe based messaging, which is designed for fast development cycles and rapid prototyping in research laboratories. The messaging latency of the system is fast enough to support latency critical applications, such as user interaction and VR projects. A case study in which we implemented sensor fusion with multiple Microsoft Kinect 2 skeleton sensors for interactive projects in VR showed that the infrastructure can improve the development process.

To improve the development process based on this platform we plan to integrate additional features based on a complex event processing (CEP) [8] engine to handle the context information and to allow the implementation of new behavior based on context changes with CEP queries. It has been shown that centralized and distributed CEP engines can improve the context handling in systems for smart environments and Internet of Things applications [9], but the most difficult part of the architecture design, will be the seamless integration into this kind of system.

Moreover, this feature should help with the analysis of system events and system state during the whole development process. This allows the developer to obtain an understanding of the system structure and potential errors more quickly with CEP Queries that filter the agent graph and show aggregated information about a subsystem. It is planned to build this features into development tools and to generalize it to provide an integrated development environment (IDE) for agent based smart environments. This IDE should improve the system monitoring, debugging and programming of new components.

References

[1] D. J. Cook, "Multi-agent smart environments," *J. Ambient Intell. Smart Environ.*, vol. 1, no. 1, pp. 51–55, Jan. 2009.

[2] G. Cugola and H.-A. Jacobsen, "Using publish/subscribe middleware for mobile systems," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 4, pp. 25–33, Oct. 2002.

[3] J. Ellenberg, B. Karstaedt, S. Voskuhl, K. von Luck, and B. Wendholt, "An environment for context-aware applications in smart homes," in *International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, 2011.

[4] M. Resnick, "All i really need to know (about creative thinking) i learned (by studying how children learn) in kindergarten," in *Proceedings of the 6th ACM SIGCHI Conference on Creativity &Amp; Cognition*, ser. C&C '07. New York, NY, USA: ACM, 2007, pp. 1–6.

[5] K. Henricksen, J. Indulska, and T. Mcfadden, "Middleware for Distributed Context-Aware Systems," in *Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems - Volume / Part I*, 2005, pp. 846–863.

[6] M. Meehan, S. Razzaque, M. C. Whitton, and F. P. Brooks, Jr., "Effect of latency on presence in stressful virtual environments," in *Proceedings of the IEEE Virtual Reality 2003*, ser. VR '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 141–.

[7] E. Langbehn, T. Eichler, S. Ghose, K. von Luck, G. Bruder, and F. Steinicke, "Evaluation of an omnidirectional walking-in-place user interface with virtual locomotion speed scaled by forward leaning angle," in *Proceedings of the GI Workshop on Virtual and Augmented Reality (GI VR/AR)*, 2015, pp. 149–160.

[8] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[9] C. Y. Chen, J. H. Fu, T. Sung, P. F. Wang, E. Jou, and M. W. Feng, "Complex event processing for the internet of things and its applications," in *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, Aug 2014, pp. 1144–1149.