Proceedings of TMCE 2020, 11-15 May, 2020, Dublin, Ireland, edited by I. Horváth and G. Keenaghan
© Organizing Committee of TMCE 2020, ISBN/EAN: 978-94-6384-131-3

INTEGRATION OF COMPLEX EVENT PROCESSING INTO MULTI-AGENT SYSTEMS: TWO USE CASES FOR DISTRIBUTED SOFTWARE DEVELOPMENT SUPPORT

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

Susanne Draheim
Kai von Luck
Department of Computer Science
University of Applied Sciences Hamburg
{ susanne.draheim, kai.vonluck }@haw-hamburg.de

Christos Grecos
School of Computing
National College of Ireland
cgrecos@staff.ncirl.ie

Qi Wang
University of the West of Scotland
School of Computing, Engineering and Physical Sciences
qi.wang@uws.ac.uk

ABSTRACT

Complex event processing (CEP) systems are typically utilised for smart environments to cope with large numbers of components and events without simultaneously increasing latencies for user interaction. Although CEP systems can simplify the development of components with CEP queries, it remains challenging to debug such complex distributed systems. Using CEP introduces an additional software development paradigm which can make it harder for developers to obtain an overview of system components or to pinpoint errors. We introduce an approach to seamlessly integrate CEP into an existing publish/subscribe middleware for smart environments, leveraging agent-based developing techniques to maintain the scalability and latency characteristics of the existing system. Our approach envisions a system in which it is irrelevant whether a feature was implemented through an agent or through a CEP query. The paper illustrates our system's architecture, describes its prototypical implementation, and exemplarily shows its applicability based on two real-world evaluations: first, a sensor network for air quality measurements; and second, an omnidirectional walking-in-place recognition system for virtual-reality applications.

KEYWORDS

Complex event processing, multi-agent systems, software development, distributed support, use cases

1. INTRODUCTION

With the latest developments in technology, smart systems have become more complex and more common. Smart technology and companion objects are integrated into a variety of environments and there are efforts to interconnect these environments to generate additional benefits for the user (Zygiaris, 2013).

These smart systems can become complex because of distributed components, heterogeneity, etc. Sensors and actors that are used in smart environments are often implemented with different programming languages and have incompatible software architectures and interfaces. Event-based architectures and middlewares are used to reduce this complexity. This can help to manage the communication of components and to achieve a loose coupling between them. Often, middleware platforms are accompanied by development tools to help with the development of new components or features (Henricksen, Indulska, & Mcfadden, 2005).

Another approach to reduce the complexity and ease

development is the integration of complex event processing (CEP) engines (Cobeanu & Comnac, 2011), that can process declarative queries (Luckham, 2001), often similar to SQL queries for databases. They enable developers to handle easier context changes and other events inside the system. Additionally, CEP queries can be used to implement entire components.

Publish/subscribe-based architectures are utilised to realise loose coupling and simultaneously provide high flexibility (Mühl, Fiege, & Pietzuch, 2006). To this end, publish/subscribe-based messaging is often utilised to implement open distributed systems.

Heterogeneous systems that include context information dependencies and high number of components can generally be challenging to debug. Integrating CEP engines increases a system's complexity, because developers have to take two different development techniques (i.e., event-based and agent-based) into account throughout their debugging activities. So even presumably easy tasks, such as finding a component which is responsible for a specific action of the system, can become a problem. In environments that are developed by many developers or by a changing team, it can be challenging to get an overview of the general system structure.

In this paper, we reflect on existing architectural designs and propose a new way to seamlessly integrate CEP into publish/subscribe-based systems for smart environments. Contrary to related designs, our approach eases development activities, such as adding components or altering a system, and simultaneously does not increase the complexity of the debugging task. To this end, our approach uses the design principles of an underlying publish/subscribe-based system.

The paper is organised as follows: first, we introduce and discuss related work; second, the paper illustrates our architecture of a seamlessly integrated CEP engine; third, we evaluate our architecture by exemplarily demonstrating its application in two case studies; finally, the paper concludes with a summary and recommendations for future work.

2. RELATED WORK

As our approach combines CEP with agent-based systems, we highlight architectures from both worlds and consider publications that combine their benefits. Additionally, we summarise different debugging techniques for agent-based and CEP systems to evaluate their applicability for the presented architectures.

2.1. Agent-based systems for smart environments

Several publications present different software architectures and agent-based middleware for smart environments or context aware applications in general (Cook, 2009).

One important feature of middleware for smart environments is the processing of context information. State or context information can be processed and stored in different ways. All information can be stored in database services. Typically, there is a central database or database layer that handles all the data. Additional services can then be used to query specific information or to inform components about context changes (Henricksen, Indulska, & Mcfadden, 2005).

Other systems either do not utilise a central database service or do not store any data at all. All information is processed live in the form of messages that are sent among the agents and the system state is held inside the agents (Novák & Dix, 2006). If data requires persistence, it is stored as needed by the responsible component. This maintains scalability but simultaneously makes it harder to debug system components, because there is no standardised way to access the state of a component. Everything has to be accessed over an application interface of the responsible component.

2.2. Complex event processing

CEP can be used to analyse event streams and recognise complex event patterns (vgl. Luckham, 2001). Complex event patterns can, for example, consist of multiple events that have to happen in a specific order. Moreover, event groups or the absence of specific events can be monitored by complex event queries.

Paschke and Vincent (2009) presented a reference architecture for CEP engines that is compatible with most event processing solutions. There are currently multiple CEP engines available. Examples are ESPER (EsperTech, 2019) and Siddhi (Suhothayan, et al., 2011). Modern CEP implementations often focus on providing high scalability with low message latency, which makes them very useful for context-based applications, smart environment sensor networks (Dunkel, 2009), and Internet of Things (IoT)

applications (Chen, et al., 2014). Furthermore, CEP engines are also used in context-aware computing in manufacturing (Alexopoulos, Sipsas, Xanthakis, Makris, & Mourtzis, 2018).

For our approach, we use the reference architecture and related CEP implementations to identify common query features and CEP engine components.

2.3. Integration of CEP into agent-based systems

Another approach to support the processing of context information in smart systems is the integration of a CEP service. This is often achieved by the integration of an existing CEP engine into a message-based middleware. There are multiple publications that combine CEP and agent-based or event-based systems in different ways.

SAGE is an agent-based monitoring and control system which is embedded in a publish/subscribe architecture (Broda, Clark, Miller, & Russo, 2009). It uses Prolog unification technology to detect complex events.

Cobeanu & Comnac (2011) presented a traffic control application that combines the JADE agent development environment (Bellifemine, Poggi, & Rimassa, 2001) with the Esper CEP engine (EsperTech, 2019). All messages in that system are routed over the CEP engine. The CEP engine can temporarily store some messages to act as a database.

However, there are certain issues with this approach. It is challenging to integrate an existing CEP engine, because the messaging layer is the most important part of an event-based system. Message throughput and the overall scalability of the system can easily be hampered by an engine that provides insufficient scalability.

A further issue is that the complexity of the system is shifted inside the CEP engine, which is often implemented with other design paradigms in mind. This makes debugging activities particularly challenging, as errors are potentially hidden inside the engine and debugging with existing debugging tools is not possible.

Multiple CEP engines can be utilised to increase the scalability and flexibility of a system (Paraiso, Hermosillo, Rouvoy, Merle, & Seinturier, 2012), but these approaches further increase the complexity.

Omicini, Fortino & Mariani (2015) propose a conceptual framework to combine abstractions and technologies from event-based and multi-agent systems which can be used as a foundation for complex software systems. There are three steps to successfully combine these systems (Mariani & Omicini, 2015). First, all agents have to be able to work as event sources and sinks; second, the systems need a uniform event model, which can be challenging to achieve in a heterogeneous system; and finally, there has to be an event-based coordination which handles the message flow for event-based and agent-based communication.

2.4. Debugging techniques

With the increasing demand and complexity of smart environments, effective debugging techniques are becoming more important.

There are many techniques and tools to support debugging in message-based distributed systems, for example, utilising event-based models of behaviour (Bates, 1995). Message tracers can be utilised to debug the message flow in an agent-based system (Bosse, Lam, & Barber, 2006).

Tools for actor-based programming, for example, model checkers (Fredlund & Svensson, 2007) or custom interactive debugging tools (Higashino, et al., 2013), (Shibanai & Watanabe, 2017) are often used. Additionally, formalised debugging techniques are available to evaluate these approaches (Torres Lopez, Boix, Scholliers, Marr, & Mössenböck, 2017).

Our approach of a seamless integration of a CEP engine allows developers to use these techniques for all parts of the system, including features that are implemented through CEP queries.

Furthermore, there are debugging techniques specifically designed for complex event processing and rule-based systems (Cugola, Margara, Pezzè, & Pradella, 2015) which enable developers to automatically check rulesets against correctness properties.

3. SYSTEM DESIGN AND IMPLEMENTATION

Our proposed architecture to seamlessly integrate CEP can be separated into three different layers based on their abstraction levels and interfaces (see Figure 1). All agents and systems components are at the bottom layer and are executed either inside runtime environments or standalone. At this layer, there is no communication between the components, but each

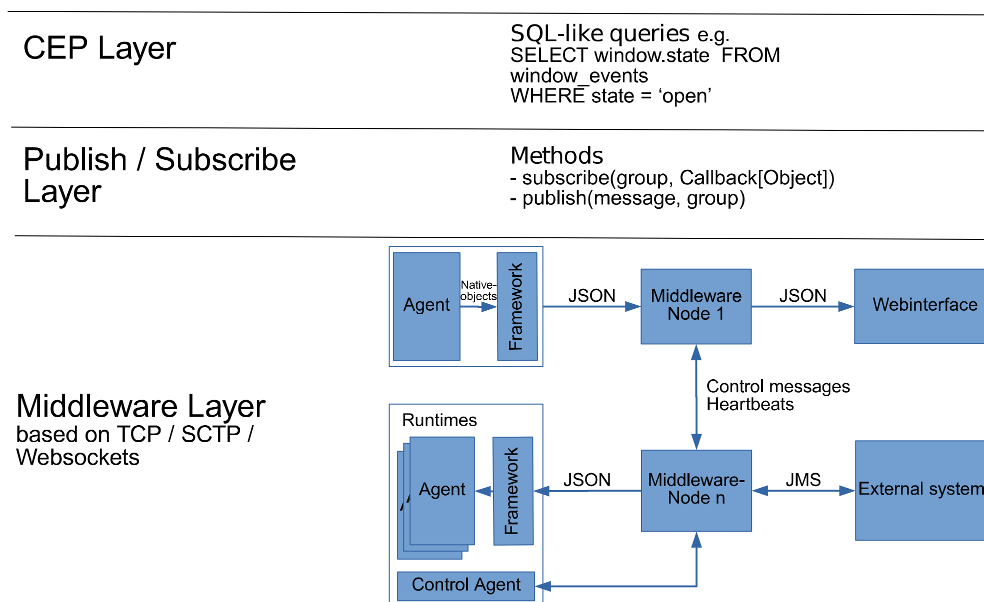

Figure 1. The three layers of abstraction inside the system after the integration of a CEP engine. Each layer is implemented on the basis of the underlying layers. The system is based on an existing middleware layer **(Eichler, Draheim, Grecos, Wang, & Luck, 2017)**

component can interact with external systems such as sensors, actors, or databases.

Each agent provides a message-based application interface that can be used to interact with the agent and use it as a service. The middleware provides a publish/subscribe interface that allows agents to send and receive messages. This architecture shapes the publish/subscribe layer on top of runtime environments.

The CEP layer is at the top. All components that form the CEP engine have to be implemented on the basis of the design principles of the existing system to seamlessly integrate complex event processing into a message-based system. Thus, we approach the design of the CEP engine as such that all queries can be executed on the basis of agent groups and that all new components have to exclusively use the messaging services to communicate with each other and other parts of the system. It uses the bottom layers to provide the interface which allows it to send, execute, and manage CEP queries. All components that provide this layer are implemented as agents which run at the bottom layer inside runtime environments. The implications are two-fold: first, layers and systems are becoming interchangeable; second, debugging tools can be used for all layers, including the CEP layer.

This seamless integration is not achievable with the integration of an existing CEP engine, such as Esper (EsperTech, 2019), because all messages have to be routed to and processed by the CEP engine.

Our CEP integration can be used with a variety of existing systems. A crucial requirement is that there is a topic-based publish and subscribe service (Baldoni, Contenti, & Virgillito, 2003) that can be used to send and receive messages from arbitrary topics. Additionally, all message values have to be accessible by a unique key such as in JSON or XML. Furthermore, the underlying system should be agent-based and provide development and debugging tools for agents to take full advantage of our approach.

We implement and test our approach based on (Eichler, Draheim, Grecos, Wang, & Luck, 2017), an agent-based middleware with publish/subscribe messaging. It is implemented using the Akka Framework and Scala programming language.

It uses a binary or text message format based on JSON which allows easy communication with external systems, such as sensors, actors, or other IoT devices. The middleware provides runtime environments to execute arbitrary agents on multiple nodes with little overhead and manages them to provide fault tolerance and optimise system performance. The system decides where an agent is initially located and can move agents to other nodes when necessary. This is used in our implementation to balance the load and minimise the latency of messages by grouping clusters of

components that heavily communicate with each other.

The CEP layer consists of three components that can create and manage other agents to implement CEP queries. The first component is the Query Parser and Optimiser, which process queries. The second component is the Query Agent Manager that creates agents to execute a query and the last component is the CEP Manager which manages all running queries. All components are shown in Figure 2 and are explained in the following sections.

3.1. Query language

We use a simple query language that is very similar to SQL. Additional languages can be added to the system by adding a component that compiles a query in the new language to a compatible abstract syntax tree (AST). Furthermore, additional language features can be added, if necessary. This can be achieved without altering or negatively affecting the rest of the system.

```
SELECT
        1 as constant,
        group_a.id as id,
        max(group_a.val1, group_b.val1)
        as max_val
INTO output_group
FROM first_input_group as group_a
JOIN second_input_group as group_b
ON group_a.id == group_b.id
WHERE group_a.id > 5
```

Listing 1. CEP query example.

The output of a query is determined by its *SELECT* clause. It defines what information is published to one or more groups. Each processing step results in at most one message per output group. Possible output values can come from input messages, functions, or constants. The *SELECT* clause can be used for a variety of cases. For example, it can be used to extract specific values from messages or apply predefined functions to them. Additionally, constants or generator functions can be used to produce new values.

The *WHERE* clause allows filtering of messages on the basis of predicates that compare fields of messages either to each other or to constants. With *JOIN* clauses, it is possible to combine multiple input messages according to a predicate. Listing 1 shows an example query that joins messages from the two groups *first_input_group* and *second_input_group* when they have an identical id value. Messages with an ID less than or equal to 5 are removed prior to the joining. If one message pair is found, the query publishes a single message containing a constant value of 1, the ID of the messages, and *max_val* set to the value of the bigger *val1* variable of the two messages. All output messages are sent to the group which is specified after *INTO* and in this case is *output_group*.

3.2. Query parser and optimiser

Every agent in the system can create new CEP queries. If a developer desires to create a query, he or she uses the provided web interface which is implemented as an agent inside the system.

Initially, the query is parsed and the syntax of the query is checked by the Query Parser, which creates an AST that is passed to the optimiser.

The Query Optimiser simplifies the AST. This is done by the elimination of unnecessary elements such as tautologies in predicates. The AST is reordered and consequently all Filter Elements are preferably at the beginning of the processing chain. This reduces the number of messages that are passed down the chain as early as possible.

After all optimisation steps are conducted, another component uses the AST to create all necessary agents. It is possible that a single agent can handle multiple graph elements. For example, a Join Element contains a filtering step. The system tries to reduce the number of agents that are created based on an AST by grouping as many elements as possible that can be processed in one step.

Finally, the system schedules the creation of all necessary agents. If there is already an agent that provides the same functionality, it is reused instead. If required, the system scales the amount of agents that are processing a step to increase the throughput of the chain. This is detected by monitoring the message inbox of each agent. In most cases, this is possible, as all manipulating or filtering of single messages are stateless and thus can be easily parallelised.

Elements that are state dependent or have to handle multiple input groups are not reusable between multiple queries, because they would change the semantic of the query. If an agent represents a bottleneck to the chain of all agents, it has to be moved to a more powerful processing node by the Query Manager.

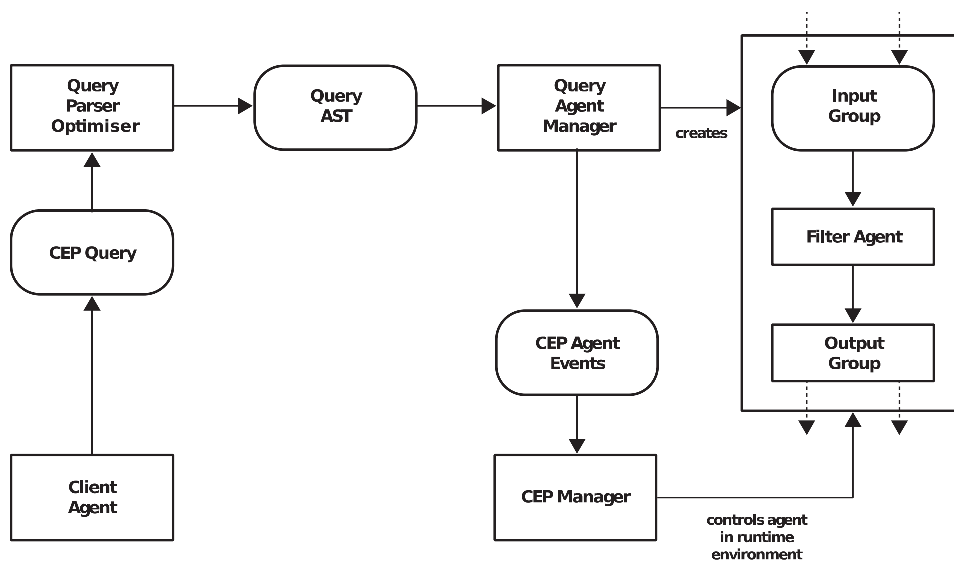

Figure 2. CEP layer components and interactions to create a new CEP query.

3.3. CEP agents

There are four different categories of CEP element agents that are required to implement all possible queries that are allowed by our query language. The language features are common elements found in other CEP implementations, as seen in Paschke & Vincent (2009) or, for example, ESPER (EsperTech, 2019).

The first type of elements are sources. These can either be an input from an existing messaging group or message generators. If an existing message group is used as an input, the first CEP element subscribes to this group.

Generators are used to create constant messages or messages with values that are generated by a function. This can be used to implement counters and random messages and provides a convenient tool for debugging purposes to generate messages.

The second type of element is used to alter messages such as Apply and Extract. These manipulate an incoming message and send it further down the processing chain. This is useful to extract specific values from messages or to apply functions to them.

Combining elements such as join are the third type of elements that are used to group messages by a predicate. This is needed to implement complex queries with multiple input channels.

Finally, there are Window elements that group messages from one source into groups of a fixed number of messages or a time window. This is required for the processing of aggregation functions such as maximum, minimum, or average.

Message sinks are not listed as a graph element, because they are not required. All graph elements can output messages to arbitrary groups. The final processing step is used to send the messages to the requested output group in the query.

- **Source**: A Source element is used to subscribe an agent element to a message group. All messages that are sent to the group are forwarded to the next elements in the graph.
- **Generator**: A Generator is utilised to generate messages based on constants or functions. This can, for example, be used to generate random messages or to provide periodic scheduled messages.
- **Extract**: An Extract element is used to extract one or more paths inside the JSON message. The values are then packaged inside a new message that is forwarded to the output stream.
- **Apply**: This element applies a given function to a message. The parameters of the function are collected from paths inside the message. An example is a $max(a, b)$ function to find the greater of two elements.

- **Join**: A Join combines two or more event streams into one. Each message that is sent to one of the input streams is forwarded to the output stream. It is possible to filter the forwarded messages based on its values. Joined messages are forwarded as tuples, with one element per input stream. All elements of the tuple can be null, depending on the type of join that is used.
- **Sliding Window**: A Sliding Window is used to combine a constant number of messages into one. It accumulates *n* messages from its input stream and forwards it as a single message to its output stream. A timeout can be used to force the forwarding of messages, even if there are fewer than *n* messages in the queue to prevent delays.
- **Sliding Time Window**: A Sliding Time Window accumulates all messages in a specific time frame. It forwards all messages that were received from the input stream grouped by their time slot as a single message to the output stream.

3.4. Agent graph

The Query Agent Manager creates an agent graph based on the AST using the CEP elements. In this step, multiple graph elements can be combined into one agent to improve the performance. For example, an element chain, where all elements have a single input and output, can be processed by a single agent. This is possible because only groups that are specified inside the *SELECT* clause are considered public to the rest of the system. Temporary groups are generated with a random identifier and are marked by the system to prevent errors.

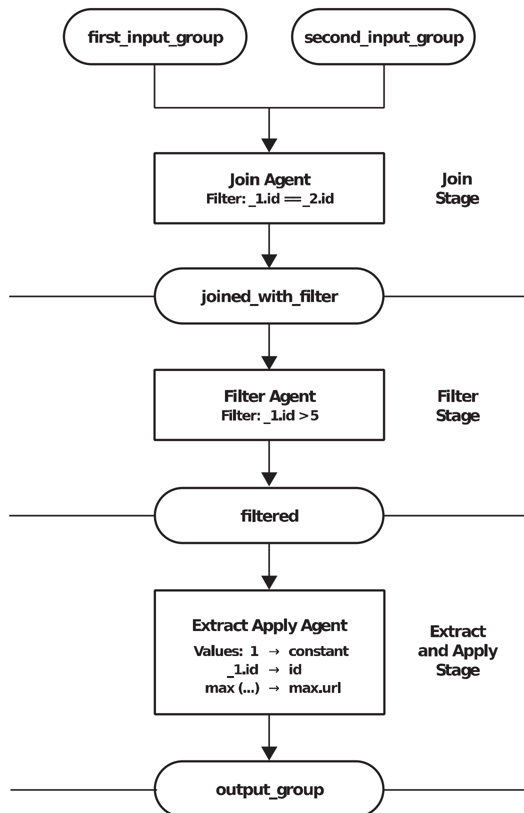

Figure 3. Graph of the example query in Listing 1.

Figure 3 shows one possible graph that is produced by the query in Listing 1. First, in the Join Stage, the two input groups are joined on the applied filter. The intermediate group *joined_with_filter* is used to pass the resulting tuples with one message from each input group with the same id. Then, in the Filter Stage, the tuples are filtered by the predicate from the *WHERE* clause of the query. Finally, the Extract and Apply Stage produces all values that were specified in the *SELECT* clause. The resulting message is then published in the *output_group*.

The number of agents that are needed to process this query is variable and depends on the decision of the Query Agent Manager. The processing of the filter, the extraction of values, and the application of functions could be separated in three stages or processed by only one. If some processing stages can be reused, it could be better to separate them. However, single-stage processing contributes the smallest latency. The implementation of the stages can be changed by the Query Manager at runtime, if necessary.

Because all agents and groups are registered to the middleware, it possible to request a list of all active entities is at any time. This can be used to visualise the state of the system, the communication flow, and the dependencies between agents.

The middleware itself is composed of multiple agents and the state of the system can be changed at any time. The entity information is provided only as a best effort to reduce the impact on the system performance to a minimum.

Consequently, after a sufficient amount of time, all agents and groups are in the entity list with their correct information. Nonetheless in the meantime, invalid information can be shown from an old state.

All information about the system is displayed with a single graph, where the agents are represented as rectangles and the groups as rectangles with rounded edges, connected by arrows which represent the publish/subscribe relationships. As the subscription of a specific group is the sole way to interact with another

agent, all dependencies are represented by the subscribe relations. The current relative message rate is indicated by the thickness of the edges.

The visualised state of the system is updated instantly as soon as changes are recognised. For example, if an agent is started, a node is added to the graph with its current subscriptions and is automatically removed as soon as the monitoring of the platform or a runtime environment detects that it is unreachable.

In a real scenario, the number of agents and groups can become significantly big. In that case, the graph visualisation would be very hard to utilise. To counteract this the graph can be filtered by agent and/or group name. Each search can optionally incorporate a configurable number of neighbours to show agents and groups that interact with the selected part of the system.

3.5. Interaction with external systems

The presented system interacts very well with external systems. An adapter can be used to translate its interface to JSON messages that are then sent to the system to integrate an external system.

4. SYSTEM EVALUATION

4.1. Case studies

Multiple case studies were conducted to evaluate the usage of our CEP integration approach.

4.1.1. Processing of environmental sensor data

One use case that occurs in almost every smart environment is the processing of context information which is collected by various sensors. We use multiple sensors to track the air quality in a smart home setting (see. Figure 4). Air quality can, for example, be measured by percentage of carbon dioxide (eCO2) or by the quantity of particulate matter in the air (PM10).

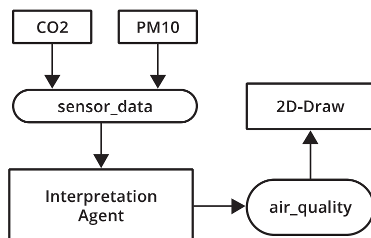

Figure 4. Overview of agents and groups to process air quality sensor measurements to automate ventilation in a smart home environment.

All air quality sensors sample every 1–3 minutes and publish their results as a message in the *sensor_data* group. The message contains an identifier of the measurement, the measurement itself, a timestamp, and the unit.

The sensor measurements are interpreted by an interpretation agent that collects all of the different measurements and outputs an air quality value between 1 and 100. The air quality value can then be used to visualise the situation to the user or to trigger actions, like opening a window, if a threshold is reached.

To test the interpretation agents and actors, which use the provided air quality data, we can use the CEP query in Listing 2. This query produces an agent chain which outputs a random value every minute into the *sensor_data* group. The messages generated by this statement are identical to messages that would come from one of the sensors. This allows the developer to use the CEP query during the development for testing. If required for testing purposes, the message rate can be easily altered in the query.

```
SELECT src.ec02
INTO sensor_data
FROM src.random.num(60000, 30, 60)
```

Listing 2. CEP query to simulate a sensor with random data in a specific range.

Alternatively, we can implement the interpretation or parts of it as CEP queries (for an example, see Listing 3). This query takes all sensor data that are generated in a 5 minute time window and sends the average of all values that are in the allowed data range to the next interpretation step.

```
SELECT avg(sensor.ec02)
INTO filtered_sensor_data
FROM sensor_data.win.time(300000) as
sensor
WHERE sensor.ec02 > 0 &&
      sensor.ec02 < 100
```

Listing 3. CEP query to aggregate all sensor data in a 300,000 ms (5 minute) time window by an average function. Values that are out of range are filtered out prior to the aggregation step.

Even if the processing of the sensor data is implemented as multiple CEP queries or agents, because of our seamless integration of the CEP processing, a developer can find the whole agent

processing chain in the agent graph. In a system with a separated CEP engine, the chain would be interrupted at all points where the engine is involved.

```
SELECT
        "CO2 Sensor Values" as name,
        sensor.name as x,
        sensor.ec02 as y
INTO draw_2d
FROM sensor_data
```

Listing 4. CEP query to visualise sensor data.

In the second step, when we implement the sensors, it is helpful to have an instant visualisation of their data (see Listing 4). This query takes all sensor measurements and transforms them into drawing commands on a 2D canvas, which results in a real-time visualisation of all past measurements. A developer can use this, for example, to test sensor implementation. As an example when he or she filters the data inside an allowed range, the corresponding event stream can be used to see the results immediately.

4.1.2. Omnidirectional walking-in-place detection

For our second case study, we use the implementation of an omnidirectional walking-in-place (WIP) detection service (Langbehn, et al., 2015). Walking in place is one method to control the movement of users inside a virtual environment. The users wear a head-mounted display and utilise their legs for movements. Sensors such as the Microsoft Kinect 2 [1] can be used to track this movement and detect a step. Each detected step is then used to perform a movement in the virtual environment. If the detection of the step and the movement are processed fast enough, it can increase the immersion for the user.

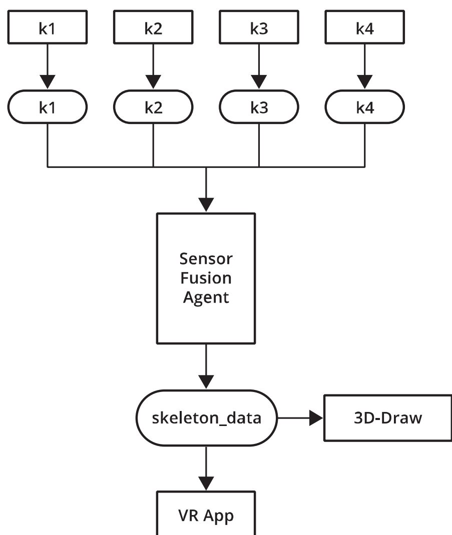

Figure 5. Overview of all agents and groups that are used to implement the omnidirectional walking-in-place detection for VR

To allow a free 360° rotation of the user, multiple sensors can be used to track the user from different angles. The sensor data from each sensor are then fused into one reliable skeleton which is used to perform the step detection. Figure 5 shows this with four sensors (k1–k4). The sensor data are collected by a sensor fusion agent and the output is then used by the VR application.

Twenty-seven joints are detected every 30 ms by each of the sensors. Even if the resulting event stream is

```
SELECT
    Vector3D(2.3, 3.4, 1.2) as head.position,
    Vector3D(2.3, 3.3, 1.1) as
spine_mid.position,
    [...]
INTO skeleton_data
FROM src.periodic(1000)
```

Listing 6. CEP query to generate skeleton data to simulate a sensor (truncated).

slowed down, it is challenging for humans to understand the three-dimensional vectors of the skeleton to debug a problem with the sensor. An easier-to-use representation would be a visualisation of the skeleton on a 3D canvas.

```
SELECT
    head.position,
    spine_mid.position,
    [...]
INTO draw_3d
FROM skeleton_data
```

Listing 5. CEP query to draw selected skeleton joints on a 3D canvas (truncated).

Listing 5 can be used to implement this on the basis of the live data from one of the sensors or the fused data. The query sends all positions of the skeleton joints to an agent that draws the points on a 3D canvas. This

[1] Kinect for Windows - https://developer.microsoft.com/windows/kinect, accessed 31.01.20

can be helpful to test the sensors or the fusing agent or to debug processing issues.

The walking-in-place detection based on the fused skeleton can also be implemented as a CEP query, as shown in Listing 7. To implement this, we take the last message from a sensor that detects the floor plane of the room and combine it with the data from one sensor. Because the rate of the skeleton data is much higher than the data from floor plane sensor, we instruct the

```
SELECT
    1 as step_detected,
    distance_point_floor(
        s.bone1.x, s.bone1.y, s.bone1.z,
        f.a, f.b, f.c, f.d
    ) as foot_height
INTO wip_events
FROM skeleton_data as s
JOIN floor_data.keep as f
WHERE foot_height > 3
```

Listing 7. CEP query to implement walking-in-place detection based on a skeleton sensor.

join operation to keep the last values, even if this joins multiple skeleton messages with one floor plane message.

We then calculate the distance between the foot joints to the floor plane. If the distance is greater than a threshold, we publish a message that indicates the step event and contains the current height of the foot.

Generally, if there is a problem with the implementation, it can result from all of the participating agents. The live representation of the system state in Figure 6 can be used to investigate such errors. Here, we can see that sensor k3 is not reachable and that sensor k4 does not send any messages. It is likely that there is a problem with both of them. As k3 is not reachable at all, it seems that the sensor hardware is defective or the agent is not operating. k4 seems operational, but the sensor does not detect the user.

The message throughput is indicated by the width of the arrows. Apparently, k2 is sending many more messages than k1. This occurs when k1 does not detect the user all the time or when sensor k2 has a higher configured frame rate.

Furthermore, the higher frame rate of k2 does not increase the frame rate of the fusion agent. It waits for data from all sensors that provide data in a configured time frame before its fuses them together. Hence, it produces a fused skeleton each time a message from k1 arrives.

This representation does not change whether the processing steps are implemented by CEP queries or manually as agents. This is only possible because the CEP integration is implemented based on the underlying system.

4.2. Latency and scalability

Systems for smart environments have to handle a large number of agents, groups, and messages, with a latency that is suitable for user interaction.

The latency of messages that are processed by a CEP query depends on the latency of the publish/subscribe-system that is utilised. Each step in the processing chain that results from a CEP query publishes the message and the next agent then receives it by its subscription. Thus, when l_{b} is the latency of the base system and n is the number of processing steps, the latency of a message is $l_{\mathrm{b}} * n + c$, where c is the sum of the overall processing time a message takes inside each agent in the processing chain.

If the latency of the implementation is compared to another implementation with the same processing steps, c becomes irrelevant, because it is identical in both systems.

The number of agents needed to implement a CEP query could be decreased by executing multiple steps in one agent, but this would violate the design principle of an agent-based system, where each agent has one specific task.

Furthermore, this is not an issue of the CEP integration but rather a fundamental decision relating to a system's design. Although an implementation separation on multiple agents can ease the understanding and load balancing, it simultaneously increases the latency.

For each CEP query, our implementation creates only a few agents and groups. Multiple queries can share parts of the agent processing chain to reduce the number of components.

Agent-based systems can easily handle a large number of components. If the underlying system is designed to scale with an increasing number of agents and groups, our CEP integration does scale at the same time and does not affect other parts of the system. This of course occurs only if the execution of agents is capsuled and managed by the base system.

5. CONCLUSION AND FUTURE WORK

Architectures that can cope with the increasing complexity of smart environment systems are becoming more relevant. One approach to building platforms for smart environments is to combine event-based and agent-based architectures to create a flexible and loosely coupled system that can handle context information.

In this paper, we present a new software architecture that seamlessly integrates CEP into an existing agent-based middleware for smart environments. We use the publish/subscribe messaging interface of an underlying system to implement an agent-based CEP engine which interacts seamlessly with all other parts of the system. This enables developers to use agent-based debugging techniques for all system components, to pinpoint specific components, or to analyse message flows. Additionally, debugging techniques for CEP queries and rule-based engines can be utilised for features that are implemented with the CEP engine.

We conducted two case studies based on the presented architecture and demonstrated its applicability for typical development tasks in smart environments. The first example was a system that interprets air quality sensor data and the second use case was an omni-directional walking-in-place detection for virtual reality applications. Furthermore, we demonstrated that our seamless CEP integration can support simple debugging tasks during the development process.

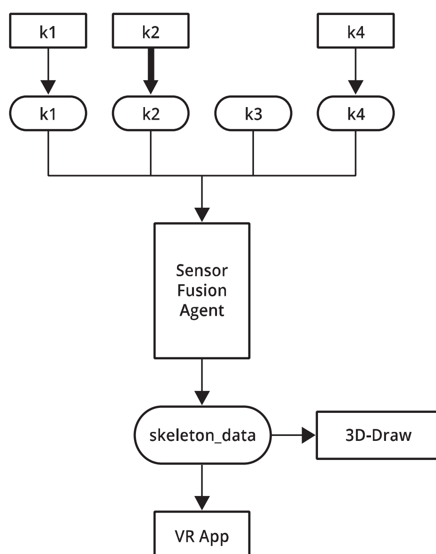

Figure 6. Possible agent graph with message throughput indicated by arrow width. This setting can help more easily debug the system.

Our architecture is a first stepping stone towards an integrated development environment for agent-based smart environments, where CEP is utilised for the development of new components and system debugging.

We plan to analyse the applicability of our system in further real-world scenarios. We will conduct a study with a larger group of developers, foremost computer science students. Additionally, we plan to further scrutinise how developers interact with such integrated development environments and how they affect the velocity of development processes in such challenging environments. A first experimental setting for virtual and augmented reality based on our platform was published separately (Becker, Meyer, Eichler, & Draheim, 2019).

REFERENCES

Alexopoulos, K., Sipsas, K., Xanthakis, E., Makris, S., & Mourtzis, D. (2018). An industrial Internet of things based platform for context-aware information services in manufacturing. *International Journal of Computer Integrated Manufacturing* (pp. 1111-1123). Taylor & Francis.

Baldoni, R., Contenti, M., & Virgillito, A. (2003). Future Directions in Distributed Computing. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, & B. Y. Zhao (Eds.). Berlin, Heidelberg: Springer-Verlag.

Bates, P. C. (1995, 2). Debugging Heterogeneous Distributed Systems Using Event-based Models of Behavior. *ACM Trans. Comput. Syst., 13*, 1-31. doi:10.1145/200912.200913

Becker, J., Meyer, U., Eichler, T., & Draheim, S. (2019, 3). A Supernatural VR Environment for Spatial User Rotation. *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, (pp. 850-851). doi:10.1109/VR.2019.8798290

Bellifemine, F., Poggi, A., & Rimassa, G. (2001). JADE: A FIPA2000 Compliant Agent Development Environment. *Proceedings of the Fifth International Conference on Autonomous Agents* (pp. 216-217). New York, NY, USA: ACM. doi:10.1145/375735.376120

Bosse, T., Lam, D. N., & Barber, K. S. (2006). Automated Analysis and Verification of Agent Behavior. *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 1317-1319). New York, NY, USA: ACM. doi:10.1145/1160633.1160876

Broda, K., Clark, K., Miller, R., & Russo, A. (2009). SAGE: A Logical Agent-Based Environment

Monitoring and Control System. In M. Tscheligi, B. Ruyter, P. Markopoulus, R. Wichert, T. Mirlacher, A. Meschterjakov, & W. Reitberger (Ed.), *Ambient Intelligence* (pp. 112-117). Berlin: Springer Berlin Heidelberg.

Chen, C. Y., Fu, J. H., Sung, T., Wang, P. F., Jou, E., & Feng, M. W. (2014, 8). Complex event processing for the Internet of Things and its applications. *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, (pp. 1144-1149).

Cobeanu, I., & Comnac, V. (2011, 5). Embedding of event processing into multi-agent systems decision mechanism. *2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, (pp. 105-109). doi:10.1109/SACI.2011.5872981

Cook, D. J. (2009, 1). Multi-agent Smart Environments. *J. Ambient Intell. Smart Environ., 1*, 51-55.

Cugola, G., Margara, A., Pezzè, M., & Pradella, M. (2015). Efficient Analysis of Event Processing Applications. *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems* (pp. 10-21). New York, NY, USA: ACM. doi:10.1145/2675743.2771834

Dunkel, J. (2009, 3). On complex event processing for sensor networks. *2009 International Symposium on Autonomous Decentralized Systems*, (pp. 1-6). doi:10.1109/ISADS.2009.5207376

Eichler, T., Draheim, S., Grecos, C., Wang, Q., & Luck, K. (2017, 10). Scalable context-aware development infrastructure for interactive systems in smart environments. *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, (pp. 147-150). doi:10.1109/WiMOB.2017.8115848

EsperTech. (2019). Esper - EsperTech. Retrieved from http://www.espertech.com/esper/

Fredlund, L.-A., & Svensson, H. (2007). McErlang: A Model Checker for a Distributed Functional Programming Language. *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (pp. 125-136). New York, NY, USA: ACM. doi:10.1145/1291151.1291171

Henricksen, K., Indulska, J., & Mcfadden, T. (2005). Middleware for Distributed Context-Aware Systems. *Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems - Volume / Part I*, (pp. 846-863). doi:10.1007/1157577153

Higashino, M., Osaki, S., Otagaki, S., Takahashi, K., Kawamura, T., & Sugahara, K. (2013). Debugging Mobile Agent Systems. *Proceedings of International Conference on Information Integration and Web-based Applications & Services* (pp. 667:667--667:670). New York, NY, USA: ACM. doi:10.1145/2539150.2539261

Langbehn, E., Eichler, T., Ghose, S., Luck, K., Bruder, G., & Steinicke, F. (2015). Evaluation of an Omnidirectional Walking-in-Place User Interface with Virtual Locomotion Speed Scaled by Forward Leaning Angle. *Proceedings of the GI Workshop on Virtual and Augmented Reality (GI VR/AR)*, (pp. 149-160).

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

Mariani, S., & Omicini, A. (2015, 5). Coordinating Activities and Change. *Eng. Appl. Artif. Intell., 41*, 298-309. doi:10.1016/j.engappai.2014.10.006

Mühl, G., Fiege, L., & Pietzuch, P. (2006). *Distributed Event-Based Systems.* Berlin, Heidelberg: Springer-Verlag.

Novák, P., & Dix, J. (2006). Modular BDI Architecture. *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 1009-1015). New York, NY, USA: ACM. doi:10.1145/1160633.1160814

Omicini, A., Fortino, G., & Mariani, S. (2015). Blending Event-Based and Multi-Agent Systems Around Coordination Abstractions. In T. Holvoet, & M. Viroli (Ed.), *Coordination Models and Languages* (pp. 186-193). Cham: Springer International Publishing.

Paraiso, F., Hermosillo, G., Rouvoy, R., Merle, P., & Seinturier, L. (2012, 9). A Middleware Platform to Federate Complex Event Processing. *2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, (pp. 113-122). doi:10.1109/EDOC.2012.22

Paschke, A., & Vincent, P. (2009). A Reference Architecture for Event Processing. *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems* (pp. 25:1--25:4). New York, NY, USA: ACM. doi:10.1145/1619258.1619291

Shibanai, K., & Watanabe, T. (2017). Actoverse: A Reversible Debugger for Actors. *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (pp. 50-57). New York, NY, USA: ACM. doi:10.1145/3141834.3141840

Suhothayan, S., Gajasinghe, K., Loku Narangoda, I., Chaturanga, S., Perera, S., & Nanayakkara, V. (2011). Siddhi: A Second Look at Complex Event Processing Architectures. *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments* (pp. 43-50). New York, NY, USA: ACM. doi:10.1145/2110486.2110493

Torres Lopez, C., Boix, E. G., Scholliers, C., Marr, S., & Mössenböck, H. (2017). A Principled Approach

Towards Debugging Communicating Event-loops. *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (pp. 41-49). New York, NY, USA: ACM. doi:10.1145/3141834.3141839

Zygiaris, S. (2013, 6 01). Smart City Reference Model: Assisting Planners to Conceptualize the Building of Smart City Innovation Ecosystems. *Journal of the Knowledge Economy, 4*, 217-231. doi:10.1007/s13132-012-0089-4