A Programming Model for Context-Aware Applications in Large-Scale Pervasive Systems

Sanjin Sehic, Fei Li, Stefan Nastic, and Schahram Dustdar
Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
Email: {s.sehic, f.li, s.nastic, s.dustdar}@infosys.tuwien.ac.at

***Abstract*—**

In recent years, new business and research opportunities have been increasingly emerging in the field of large-scale context-aware pervasive systems (e.g. pervasive health-care, city traffic monitoring, environmental monitoring, smart grids). These large-scale pervasive systems are characterized by the need to employ large number of context sources, process massive amounts of real-time context data, provide services to numerous context-aware applications, and cope with higher volatility of the environment.

This paper proposes the Origins Model — a programming model for context-aware applications in large-scale pervasive systems. In the Origins Model, an origin is an abstraction of any source of context information. Origins are universal, discoverable, composable, migratable, and replicable components that are associated with type and meta-information. They create an adequate foundation for the development of context-aware applications. Based on them, four processing operations are defined in the Origins Model: filter, infer, aggregate, and compose. As such, these operations provide a powerful mechanism to express a rich set of processing schemes in context-aware applications. Based on the Origins Model, we present the Origins Toolkit — a proof-of-concept implementation developed using the Scala programming language and the Akka toolkit to provide a distributed, scalable, and fault-tolerant solution.

I. Introduction

Context-awareness is one of the cornerstones of pervasive computing [1, 2]. It refers to the idea that an application can understand its context, reason about its current situation, and perform suitable operations based on this knowledge. Moreover, as the situation changes over time, the application should adapt its behavior according to new circumstances. In pervasive systems, context information is gathered from numerous heterogeneous context sources. These sources are mostly low-level sensors that are unaware of application requirements and context information acquired from such sensors is too fine-grained and low-level for context-aware applications to consume. Thus, the task of pervasive systems is to hide the complexity of acquiring context information from context sources, allow this information to be processed to create higher-level context information, and provide context-aware applications with an easy interface to retrieve the context information and adapt to its changes accordingly.

Recent years showed emerging trends in business and research to utilize large-scale pervasive systems. Examples of such trends are pervasive health-care, city traffic scheduling, environmental monitoring, and smart grids. These systems differ significantly from conventional context-aware systems, which focus on a limited personal context in relatively controlled environments (e.g. smart homes and offices). Such large-scale pervasive systems are characterized by the need to employ large number of context sources in acquisition of context information and, consequently, have to process massive amounts of real-time context data for numerous and diverse context-aware applications. Furthermore, large-scale pervasive systems also need to cope with higher volatility of the environment. Context sources are connected through geographically distributed, heterogeneous, and unreliable networks. Failures and unexpected delays to acquire context information are inevitable. It is imperative for such large-scale pervasive systems to have a dedicated programming model that transparently deals with these challenges and offers context-aware applications with a scalable solution to obtain and process context information.

This paper presents the Origins Model and the Origins Toolkit. The Origins Model is a programming model for the development of context-aware applications in large-scale pervasive systems. Its design allows a large-scale pervasive system to provide a flexible infrastructure and to easily scale with the increase in the number of context-aware applications and the volume of data that they require. The core idea in the Origins Model is that *origins* provide an adequate abstraction to represent any type of context source like sensors, web services, databases, files, and even compositions of other origins. They are universal, discoverable, composable, migratable, and replicable components that are associated with type and meta-information and create an elementary building block in the development of context-aware applications. Based on the origins, four processing operations are defined in the Origins Model, namely *filtering*, *inference*, *aggregation*, and *composition*, that further support the development of context-aware applications. They provide a powerful mechanism to express a rich set of processing schemes by using either a single one of them or a composition thereof. Based on the Origins Model, we present the Origins Toolkit as a proof-of-concept implementation developed using the Scala program-

978-1-4673-1430-5/12/$31.00

ming language[1] and the Akka toolkit[2]. The implementation uses *actors* [3, 4, 5], *futures* [6], and *promise pipelining* [7] to provide a distributed, scalable, and fault-tolerant solution.

The paper is structured as follows: the related work is surveyed in Section II. The Origins Model and key concepts behind the programming model are introduced in Section III. Section IV presents the Origins Toolkit, an implementation of the Origins Model. Finally, Section V concludes the paper with the future work.

II. Related Work

Due to the usual scale of pervasive systems and context-aware applications, past research studies in the field have not been focused on creating a dedicated programming model (i.e. an abstraction) to ease the understanding and development of large-scale pervasive systems and context-aware applications.

The Technology for Enabling Awareness (TEA) project and its successors aimed at augmenting mobile devices and everyday environments with context-awareness. The TEA architecture is a layered architecture that can synthesize context information from a heterogeneous set of sensors [8]. Similarly, the Context Toolkit provides a programmatic support for the development of context-aware applications [9, 10]. The Context Toolkit incorporates various services related to gathering and provisioning of context information, including encapsulation of context, access to context information, storage, and a client-server infrastructure. Both the TEA and the Context Toolkit are examples of the context-aware programming models for building embedded context-aware applications typically using sensors on different devices. Very little support for the distributed environmental sensing was provided. For example, obtaining meta-information about sensors (e.g. sensor's location) in the environment was not directly addressed.

More distributed frameworks and architectures for context acquisition, management and usage were examined in the Pervasive, Autonomic, Context-aware Environment (PACE) project and in the Java Context-Awareness Framework (JCAF). The PACE infrastructure aims at facilitating the development of context-aware applications through provisioning of generally required programming functionality like context gathering, context management, and context dissemination [11]. JCAF is a Java-based service-oriented run-time infrastructure and API for the creation of context-aware applications [12]. The JCAF run-time infrastructure emphasizes security and privacy in an environment of distributed and cooperating services that acquire context information through Context Monitors and Context Actuators. It enables interested applications to subscribe to relevant context events through an event-based publish-subscribe mechanism. Although both solutions provide distributed programming models, they were targeting relatively contained domains, such as smart homes and smart offices, and did not provide any mechanisms to scale their solutions pragmatically. For example, after the system is deployed, it is very hard to extend it with new computational resources to cope with an increased number of sensors.

More recently, there has been research in developing a broker-based programming model for context-aware systems [13, 14, 15]. For example, the Context Casting (C-CAST) [15] project proposed a broker-based context-provisioning system that is supported by the publish-subscribe mechanism. Components in C-CAST take role of either a context provider or a context consumer. The task of the context broker is to hold registrations of all context providers and offer it as a directory service for context consumers. The biggest concern with the broker-based programming model is the centralization of the knowledge of the whole system in the context broker. The context broker mediates almost all communication between context consumers and producers, which inherently limits the ability of the system to scale.

In our previous research, we developed the COPAL middleware [16] and provided a macro language for rapid development of context-aware applications [17]. The COPAL middleware provides a run-time infrastructure for the context provisioning and was part of the smart homes infrastructure developed in the SM4ALL project[3]. Context provisioning refers to the approach of acquiring, processing and disseminating context information in order to raise context-awareness in applications. Despite having a distributed programming model based on complex event processing, the concepts defined in the COPAL middleware also did not scale well beyond the context-aware pervasive systems of the size of smart homes.

In summary, prior research studies in programming models for development of context-aware applications were suitable only for small and relatively confined environments like smart homes or offices. Not much effort was put into allowing systems and programming models to scale beyond these sizes. As such, there is a significant gap between the vision of the context-awareness in pervasive computing from [1, 2] and the current state-of-the-art with respect to size and scalability of previously proposed context-aware programming models.

III. The Origins Model

The Origins Model (Fig. 1) is a programming model that provides a flexible abstraction of large-scale pervasive systems. The main driving force in the design of the Origins Model was to ease the development of context-aware applications. Moreover, its design allows large-scale pervasive systems to gracefully scale up with respect to increase in the number of context-aware applications and context sources.

The primary idea behind the design of the Origins Model is that all context sources can be defined as *origins*. More formally, an origin is an abstraction of a single context source in pervasive systems. Thus, origins define elementary components in the development of context-aware applications.

A. Origin's Design

In pervasive systems, any information that can be acquired and is relevant in describing the environment is considered

[1] http://www.scala-lang.org/

[2] http://akka.io/

[3] http://www.sm4all-project.eu

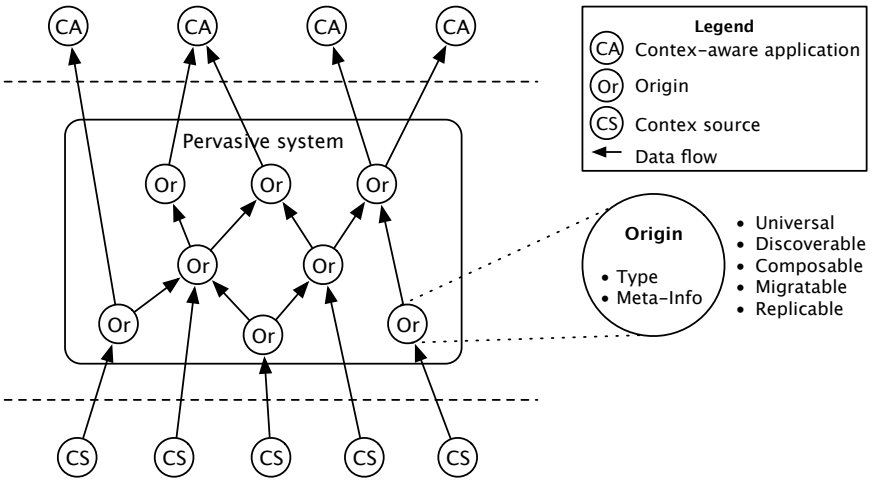

Fig. 1. The Origins Model

context information [18]. In the Origins Model, origins provide an *universal* interface for context-aware applications to access context information. They acquire the context information from any source that can provide them with information about the environment like sensors, web services, files, databases, and even compositions of other origins. The actual mechanism that an origin uses to acquire context information is irrelevant for context-aware applications and is therefore hidden. This allows context-aware applications to be built without needing to know how some particular context information is being sensed.

In the Origins Model, each origin is associated with a *type* (e.g. temperature, location, presence) that describes which context information it provides. The type of an origin is associated with the origin during its creation and remains unchanged during its lifetime. Furthermore, origins are associated with *meta-information* about the context information they provide. This meta-information allows context-aware applications to further filter relevant origins from irrelevant ones. Examples of meta-information are a geographical region with which context information is associated (e.g. temperature in Vienna), security and privacy policies (e.g. encryption methods and public keys), and so forth. The meta-information can be set internally by the origin or externally by clients and is allowed to change during the lifetime of the origin. For example, the location of a mobile sensor is not static and changes as the sensor is moved. In a pervasive system, origins associated with specific type and meta-information must be *discoverable* by context-aware applications.

Context-information acquired by origins can be used to produce new types of context information. For example, context information can be created from another context information (e.g. conversion of temperature from Celsius to Kelvin), an aggregation of other context information (e.g. summing electrical consumption of all appliances to provide current overall consumption in a house), or even a deduction from the presence or absence of some other context information (e.g. using location of people to provide the occupancy of a room). In the Origins Model, an origin is allowed to use other origins as context sources with the intent to produce new context information. For example, we can have an origin that provides an average electrical consumption of some particular household appliance like fridge, air-conditioning, or television in some particular region. This context information has to be determined from origins that provide current electrical consumption of such appliance and are located in specified region. This mechanism of *composing* origins in any particular fashion provides a very powerful method to implement processing schemes for context information.

In the design of the Origins Model, origins can be physically separated from their context sources. This separation allows origins to be *migratable* and executable on a different machine than the context source or any context-aware application. Furthermore, a context source can have multiple origins representing it and if these origins share context information and meta-information between them, then origins are also *replicable*. These two properties of origins, allow large-scale pervasive systems, that implement the Origins Model, to easily scale and adapt to change in the number of context sources (migration) and the load that context-aware applications put on origins (replication).

More formally, origins have to exhibit these properties:

1) *Universal:* An origin is an abstraction of a single context source that provides an universal interface for context-aware applications to access its context information.
2) *Discoverable:* Origins are discoverable based on the type of context information they provide and meta-information associated with them.
3) *Composable:* Origins can be composed into another origin that provides a new type of context information, which is produced from context information of the composed origins.
4) *Migratable:* Origins can migrate between different machines to improve flexibility and scalability.

5) *Replicable:* Multiple origins can abstract the same context source to improve reliability, fault-tolerance, and accessibility.

B. Operations

Context-aware applications use context information to reason about their current environment and provide relevant services to their users by adapting to changes in the environment [18]. When developing a context-aware application, developers have to answer three questions:

1) Which context information is relevant for the application?
2) How can the application understand the obtained context information with respect to the environment?
3) How to adapt application's behavior based on the new information about the environment?

Thus, the task of pervasive systems is to allow context-aware applications to *access* relevant information, to *process* the received information, and to *react* to changes in the environment.

In the Origins Model, two operations, *select* and *retrieve*, are sufficient to access context information provided by origins.

1) *Select* operation allows context-aware applications to discover origins that have specific type and meta-information (i.e. adhere to specific criteria).
2) *Retrieve* operation allows context-aware application to acquire current value of context information from an origin.

We should note that select and retrieve operations can return no result because for example the selection criteria was too strict or there was a problem with retrieving the current value of context information (e.g. network problem). In small-scale pervasive systems, it is allowed and even encouraged to raise an exceptional state when there is a problem with accessing context information. Such pervasive systems have limited number of context sources. Misbehavior of a context source should be immediately noticed and dealt with, because the loss of even one context source can create a negative impact on the functionality of the whole system. In large-scale pervasive systems, we have tens of thousands of context sources distributed over a large area and connected over an unreliable network. Problems with accessing context sources become a common occurrence. Furthermore, such pervasive systems have much bigger redundancy with respect to context sources and can easily cope with failure of one by accessing another source. Thus, the failure to access context information should not be considered as an exceptional state. Nevertheless, the failure is noted by returning no result and providing a possible reason for it.

The two simple operations for accessing context information are enough to build any type of context-aware application, but it is helpful to have additional processing operations to ease the development of more complex applications by allowing to explicitly specify processing of context information. In the Origins Model, each processing operation is implemented on top of the select and retrieve operations. We can distinguish between four different processing operations (Fig. 2): *filter* (1:1), *infer* (n:1), *aggregate* (1:n), and *compose* (n:m). The number in parentheses represents the ratio between the number of different types of the context information used as input for the processing operation and the number of distinct values of context information generated as output by the processing operation.

1) *Filter:* Filtering lets programmers specify further pruning of unneeded context information based on their current values. A filter operation defines criteria for inclusion of a retrieved value of context information into the result. This is different from the selection operation, which defines criteria for inclusion of context sources. This operation can be implemented on top of the retrieve operation. When the retrieve operation returns a current value of context information, the filter operation checks the result against the criteria and if the result passes the criteria, it is returned. Otherwise, no result is returned.
2) *Infer:* Inference is a composition of heterogeneous context information into one new context information. A simple example of inference is an operation that uses air temperature and relative humidity to determine the heat index. This operation is implemented by first retrieving current values of context information from multiple origins, and then invoking the inference algorithm on the values and returning its result.
3) *Aggregate:* Aggregation is defined as a process of combining multiple values of the same context information into a homogeneous list. This type of information can be beneficial as a source to analyze deviations between the values in the list or to infer new information from the values (e.g. average, sum, minimum, maximum). Aggregation can be separated into two distinct types: *scope-based* and *time-based*. The scope-based aggregation collects context information from all available origins that satisfy some selection criteria (e.g. all temperatures in Austria per city). It is implemented by invoking the select operation and returning all current values of context information from all origins returned by the select operation. The time-based aggregation collects context information from a single origin in some predefined time period (e.g. temperature every 10 seconds in next 5 minutes). It is implemented on top of the retrieve operation by repeatedly invoking it based on the specified rate and period.
4) *Compose:* Composition allows context-aware applications to create a Cartesian product of multiple, heterogeneous context sources. As this can create an exponential number of resulting values, it is almost always necessary to include a filtering step that specifies which products are considered "valid". An example of this operation is a composition of the current temperature and atmospheric pressure (the product part) when they come from the same location (the filtering part). This operation is implemented using multiple selection criteria to select origins

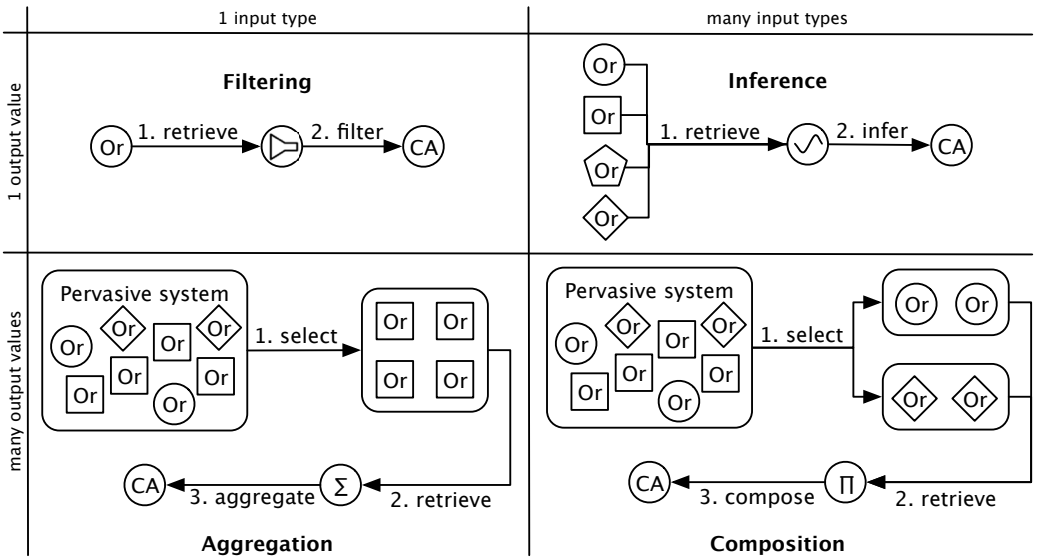

Fig. 2. Processing Operations

and retrieve current values of context information from all origins in each selected group, and then creating a Cartesian product of current values between each selected group of origins.

Also, we should note that processing operations are *composable* with each other. This allows context-aware applications to define processing schemes by using either the four elementary processing operations or a composition thereof.

Lastly, the Origins Model defines one more helpful operation, *monitor*, that a context-aware application can use to react to changes in context environment.

1) *Monitor:* The monitoring of context information allows programmers to specify periodic retrieval of context information and to associate a task that is executed on each retrieval. Thus, it provides functionality to design applications that continuously monitor context information and asynchronously invoke its business logic as the environment changes.

IV. The Origins Toolkit

The Origins Toolkit is our implementation of the Origins Model. It is implemented in the Scala programming language using the Akka toolkit. Scala is a general-purpose programming language, which integrates features of object-oriented and functional programming paradigms. It runs on Java Virtual Machine and is byte-code compatible with Java applications. We choose to implement the Origins Toolkit in Scala because of its flexibility to easily create domain-specific languages and package them as libraries. The Akka toolkit is an implementation of the Actor Model. It provides a platform to build concurrent, scalable, and fault-tolerant applications.

The Actor Model provides the abstraction for transparent distribution of concurrent computations [5]. In the Actor Model, an *actor* is a universal primitive to represent concurrency [3]. It reacts to messages that it receives from the outside by sending other messages, creating new actors, and changing its behavior. The Origins Toolkit uses actors as foundation to implement the origins. Thus, the only way to communicate with an origin is by sending it messages and receiving replies from it. This decouples context-aware applications from the actual instances of origins and, together with the location transparency provided by the Akka toolkit, allows the implementation, deployment, and use of origins in a distributed, scalable, and fault-tolerant fashion.

In the Origins Toolkit, the retrieval of context information from origins is based on the concept of *futures*. A future represents "a promise to deliver the value" [6]. This mechanism allows context-aware applications to decide if they are going to wait until the result of the future is known (synchronous), or to continue with execution of some other task and use the result when it is available (asynchronous). Moreover, an interesting aspect of futures is that it enables the *promise pipelining*, which is used by the Origins Toolkit to implement the processing of context information with minimal latency. It allows defining a processing operation on top of a future instead of an actual value. For example, the conversion of temperature from Celsius to Kelvin is implemented by combining the future that will return a temperature in Celsius and the conversion function from Celsius to Kelvin into a new future. This future will be completed (i.e. have a value) when the temperature in Celsius from the first future is available. Thus, the concept of processing in the Origins Toolkit can be described as a method to combine futures and processing functions into new futures, which can then be further processed or used directly by context-aware applications.

The Origins Toolkit (Fig. 3) is conceptually separated into three components: `System`, `Client`, and `Origin`. `System` is responsible for creation and maintenance of origins. `Client` is associated with a `System`. It is used by a context-aware application to select origins created within the `System` and define

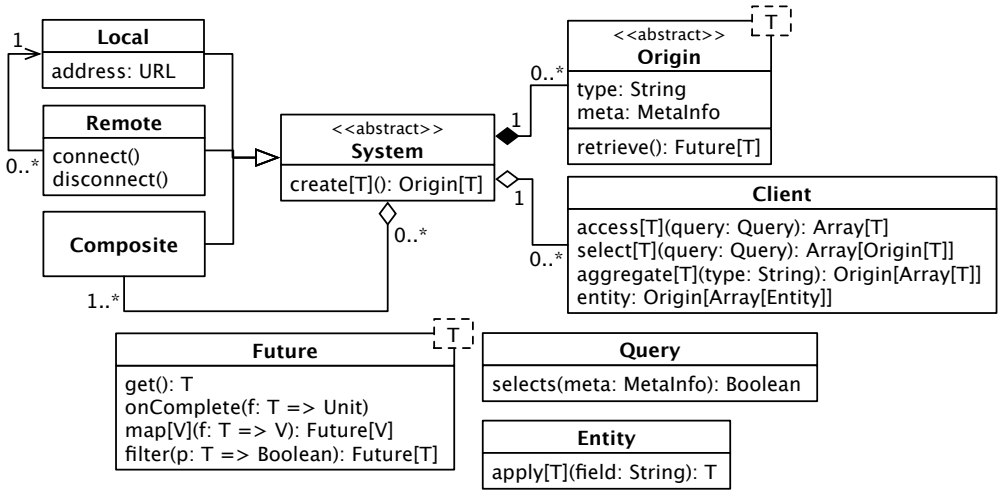

Fig. 3. The Origins Toolkit components

processing operations. Finally, `Origin` provides access to its type of context information and meta-information, and allows context-aware applications to retrieve context information.

A. System

The Origins Toolkit defines three different implementations of `System`: `Local`, `Remote`, and `Composite`.

An instance of the `Local` system creates origins locally. Thus, origins created in an instance of the `Local` system are executing in the same Java Virtual Machine as the instance of the `Local` system. Furthermore, the `Local` system exposes a remotely-accessible interface using the Akka toolkit to create and manage origins. Listing 1[4] shows an example of instantiating the `Local` system and creating a temperature `Origin`.

```
val system = new System.Local {
  origins.create("temperature") {
    // code to access temperature sensor
  }
}
```

Listing 1. A Local System with a temperature origin

An instance of the `Remote` system uses the remote interface provided by an instance of the `Local` system. Origins created with an instance of the `Remote` system are serialized as bytecode and transferred over the network to the instance of the `Local` system. This same mechanism allows the Origins Toolkit to *migrate* origins between the systems. Listing 2 shows an example of instantiating and connecting an instance of the `Remote` system.

```
val system = new System.Remote(url, 1234)
system.connect()
```

Listing 2. A Remote System

[4]All listings are expressed in the Scala programming language.

An instance of the `Composite` system composes a set of systems into one larger system. When a `Composite` system is used to create an `Origin`, it creates the `Origin` on each of the underlying systems and then instantiates a local proxy `Origin` that uses these underlying `Origins` to access context-information in a round-robin fashion. This mechanism allows origins to be *replicated* across many instances of `System`. Furthermore, the `Composite` system also exposes a remote interface, which allows it to be accessed by a `Remote` system or be part of other `Composite` systems. Listing 3 shows an example of instantiating a `Composite` system.

```
val r1 = new System.Remote(url1, 1234)
val r2 = new System.Remote(url2, 2345)
val system = new System.Composite(r1, r2)
```

Listing 3. A Composite System

B. Client

An instance of `Client` is created using an instance of `System`. It provides the `select` method to select origins created with the `System` and the `access` method that synchronously selects origins and retrieves their context information. Listing 4 shows an example of creating an instance of `Client` and accessing the temperature context information.

```
val remote = new System.Remote(url, 1234)
val client = new remote.Client
val temperatures: Array[Double] =
  client.access("temperature")
```

Listing 4. Access of temperature context information

Both the `select` and the `access` methods require an instance of `Query` as argument, which allows context-aware applications to specify the required type of context information and the meta-information that has to be associated with an origin. Most commonly, an instance of `Query` is created from

a list of key-value pairs, which represent a selection criteria based on origin's meta-information. Listing 5 shows selecting all temperature origins located in Vienna using a `Client` and retrieving future values from the selected origins.

```
val origins: Array[Origin[Double]] =
  client.select(
    "type" -> "temperature",
    "location" -> "Vienna"
  )
// the Futures.sequence method returns
// Future[Array[T]] from Array[Future[T]]
val temperatures: Future[Array[Double]] =
  Futures.sequence(
    for (origin <- origins)
      yield origin.retrieve()
  )
```

Listing 5. Selection and retrieval of context information

A future value returned by the `retrieve` method can be synchronously accessed using the `get` method or a task can be defined using the `onComplete` method, which will be executed asynchronously when the value is available. Both mechanisms are demonstrated in Listing 6.

```
// synchronous
val values: Array[Double] = temperatures.get()
// asynchronous
temperatures.onComplete {
  temperatures => // do something
}
```

Listing 6. Accessing future value of context information

Using `Client` and `Future` allows context-aware applications to define processing operations. Filtering of context information uses the `filter` method in `Future` to define a predicate function for filtering. Listing 7 shows how to filter non-positive temperature.

```
val positiveTemperature: Future[Double] =
  origin.retrieve().filter {
    temperature => temperature > 0
  }
```

Listing 7. A Filtering of context information

The inference of context information is implemented using the `map` method in `Future` to transform retrieved values of context information into a different type. Listing 8 shows how we can find a minimum temperature from an array of temperatures.

```
val minimum: Future[Double] =
  temperatures map {values => values.min}
```

Listing 8. Inference of context information

Listing 9 shows how we can define a scope-based and a time-based aggregation. It uses the `aggregate` method to create a new proxy `Origin` which does the respective aggregation of context information when its `retrieve` method is invoked.

```
// the scope-based aggregation
val origin: Origin[Array[Double]] =
  client.aggregate("temperature")
val temperatures: Future[Array[Double]] =
  origin.retrieve()

// the time-based aggregation
val origin: Origin[Array[Double]] =
  client.aggregate(
    "temperature",
    for = 5.minutes,
    every = 10.seconds
  )
val temperatures: Future[Array[Double]] =
  origin.retrieve()
```

Listing 9. Aggregation of context information

Finally, `Client` provides the `entity` method for composition of context information. The `entity` method creates a new proxy `Origin`, which does a Cartesian Product of the specified types of context information. When the `Origin`'s `retrieve` method is invoked, it returns an array of `Entities` where each `Entity` represents a single Cartesian product. Listing 10 shows how we can get the first Cartesian product of temperature and humidity in Vienna.

```
val origin: Origin[Array[Entity]] =
  client.entity.
    field[Double]("temperature" where
      {"location" -> "Vienna"}).
    field[Int]("humidity" where
      {"location" -> "Vienna"})
val entities: Future[Array[Entity]] =
  origin.retrieve()

// get first entity
val first: Entity = entities.get()(0)
val temperature: Double = first("temperature")
val humidity: Int = first("humidity")
```

Listing 10. Composition of context information

V. Conclusion

This paper introduced the Origins Model and the Origins Toolkit. The Origins Model provides a programming model for the development of context-aware applications on large-scale pervasive systems. The abstraction provided by the Origins model is based on universal, discoverable, composable, migratable, and replicable origins that are associated with type and meta-information. These properties allow origins to adequately represent many different types of context sources. Based on origins, we defined four elementary processing operations: filter, infer, aggregate, and compose. These operations provide

sufficient foundation to create processing schemes in context-aware applications. Based on the Origins Model, we developed the Origins Toolkit, which is our proof-of-concept implementation that utilizes concepts of actors, futures, and promise pipelining to provide a distributed, scalable, and fault-tolerant solution.

The Origins Toolkit provides the necessary support to scale a large-scale pervasive system with increasing number of context sources and context-aware applications using migration and replication of origins. For future work, we plan to extend the Origins Toolkit to support intelligent and elastic scaling of large-scale pervasive systems using ideas from cloud computing research. Furthermore, we plan to integrate the concept of data quality from the research into database systems into the Origins Model and Toolkit. This concept will allow a large-scale pervasive system to process and disseminate context information in a much more consistent way and provide higher-quality context data to context-aware applications.

ACKNOWLEDGMENT

This work is supported by Pacific Controls Cloud Computing Lab[5] (PCCCL)— a joint lab between Pacific Controls LLC, Sheikh Zayed Road, Dubai, United Arab Emirates and the Distributed Systems Group of the Vienna University of Technology.

REFERENCES

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 3, no. 3, pp. 3–11, February 1991.

[2] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal Communications*, vol. 8, no. 4, pp. 10–17, 2001.

[3] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[4] C. Hewitt and J. Henry C. Baker, "Laws for communicating parallel processes," in *Proceedings of the 1977 IFIP Congress*, August 1977, pp. 987–992.

[5] G. A. Agha, "ACTORS: A model of concurrent computation in distributed systems," MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA, Tech. Rep. AITR-844, June 1985.

[6] J. Henry C. Baker and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, New York, USA: ACM Press, 1977, pp. 55–59.

[7] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*. New York, New York, USA: ACM Press, 1988, pp. 260–267.

[8] A. Schmidt and K. van Laerhoven, "How to build smart appliances?" *IEEE Personal Communications — Special Issue on Pervasive Computing*, vol. 8, no. 4, pp. 66–71, 2001.

[9] D. Salber, A. K. Dey, and G. D. Abowd, "The context toolkit: Aiding the development of context-enabled applications," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: The CHI is the Limit*. New York, New York, USA: ACM Press, 1999, pp. 434–441.

[10] A. K. Dey, G. D. Abowd, and D. Salber, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications," *Human-Computer Interaction*, vol. 16, no. 2, pp. 97–166, December 2001.

[5] http://pcccl.infosys.tuwien.ac.at/

[11] K. Henricksen and J. Indulska, "Developing context-aware pervasive computing applications: Models and approach," *Pervasive and Mobile Computing*, vol. 2, no. 1, pp. 37–64, February 2006.

[12] J. E. Bardram, "The java context awareness framework (JCAF) — a service infrastructure and programming framework for context-aware applications," in *Proceedings of the 3rd International Conference on Pervasive Computing*. Munich, Germany: Springer Verlag, 2005, pp. 98–115.

[13] H. Chen, T. Finin, and A. Joshi, "Semantic web in the context broker architecture," in *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*. Washington, DC, USA: IEEE Computer Society, March 2004, pp. 277–286.

[14] M. J. van Sinderen, A. T. van Halteren, M. Wegdam, H. B. Meeuwissen, and E. H. Eertink, "Supporting context-aware mobile applications: An infrastructure approach," *IEEE Communications Magazine*, vol. 44, no. 9, pp. 96–104, September 2006.

[15] M. Knappmeyer, N. Baker, S. Liaquat, and R. Tönjes, "A context provisioning framework to support pervasive and ubiquitous applications," in *Proceedings of the 4th European Conference on Smart Sensing and Context*. Berlin, Hidelberg, Germany: Springer Verlag, 2009, pp. 93–106.

[16] F. Li, S. Sehic, and S. Dustdar, "COPAL: An adaptive approach to context provisioning," in *Proceedings of the 6th International Conference on Wireless and Mobile Computing, Networking and Communications*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 286–293.

[17] S. Sehic, F. Li, and S. Dustdar, "COPAL-ML: A macro language for rapid development of context-aware applications in wireless sensor networks," in *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*. New York, New York, USA: ACM Press, 2011, pp. 1–6.

[18] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*. London, United Kingdom: Springer Verlag, 1999, pp. 304–307.