The Trust-Enabling Middleware: Introduction and Application

Rolf Kiefhaber, Florian Siefert, Gerrit Anders, Theo Ungerer, Wolfgang Reif

Institut für Informatik
Universität Augsburg

Report 2011-10 March 2011

Institut für Informatik
D-86135 Augsburg

The Trust-Enabling Middleware: Introduction and Application

Rolf Kiefhaber, Florian Siefert, Gerrit Anders, Theo Ungerer, Wolfgang Reif

Institute of Computer Science

Augsburg University, Germany

E-Mail: {kiefhaber, siefert, anders, ungerer, reif}@informatik.uni-augsburg.de

Abstract

In this report, we present the Trust-Enabling Middleware (TEM) that is based on the message- and service-oriented organic middleware OCµ. The TEM enhances OCµ by features that enable the middleware as well as applications based on it to use trust data. These features include the possibility to save experiences made with interaction partners and to derive trust data with the help of trust metrics out of these saved experiences. Furthermore, we show an example application based on the Trust-Enabling Middleware that considers uncertainty in power networks, the Trusted Energy Grid, and especially illustrate its use of the Trust Metric Infrastructure provided by the TEM.

1 Introduction

Many distributed applications use the concept of trust as a way to deal with uncertainty or to enable cooperation among its entities. For this purpose, these entities must have the capability to save experiences made with interaction partners in a specific context, to interpret these experiences with a specific metric to trust values, and to query trust values. However, it is not useful that all applications come up with individual solutions to this generic problem. Therefore, a common foundation that provides an infrastructure for storing, interpreting, and querying trust data is required.

To be able to provide a generic infrastructure for applications that have the need for the concept of trust, we therefore developed the *Trust-Enabling Middleware (TEM)*. The TEM is based on the message- and service-oriented organic middleware OCμ [8], a middleware that uses Organic Computing principles [4] and thus incorporates self-x properties, such as the ability to configure, optimize, and heal itself. The Trust-Enabling Middleware extends OCμ by the concept of trust. On the one hand, the TEM hence provides generic functionality for applications running on top of it that need to save, interpret, and query trust related information. This functionality is encapsulated in the *Trust Metric Infrastructure*. On the other hand, the TEM can increase its robustness by making use of trust data itself by considering trust in the self-x algorithms.

Trust, as we define it [7], is a multi-faceted concept. It considers the faith of users in the system as well as the faith of entities in other entities. It thus enables cooperation among entities, like needed, for example, in multi-agent systems [5]. An important property of trust is that it depends on the given context as, for instance, you may trust your doctor in giving therapy to you when you are ill but not in repairing your car. Moreover, trust evolves over time and is a subjective concept. It consists of the facets functional correctness, safety, security, reliability, credibility, and usability.

If different facets of trust are regarded, different measures have to be taken either at design time or at run time. This report focusses on reliability and credibility. Reliability gives a measure of the availability, resilience, performance of a system, while credibility gives a measure of how beneficially an interaction partner behaved in the past. Both facets are quantifiable and can be measured and evaluated while a system is running. While reliability is measured by the middleware, credibility has to be evaluated by the application.

In this report, we describe the architecture of the Trust-Enabling Middleware and its Trust Metric Infrastructure. Moreover, we show an example application based on the Trust-Enabling Middleware. The remainder of this report is structured as follows: Sect. 2 presents the architecture of OCμ. In Sect. 3, we give a detailed introduction to the TEM's Trust Metric Infrastructure that enables applications based on the TEM as well as the TEM itself to save, interpret, and query trust data. Afterwards, an application that uses the Trust-Enabling Middleware and its use of the provided functionality, especially the Trust Metric Infrastructure, is shown in Sect. 4. Finally, Sect. 5 concludes this report.

2 Architecture

The Trust-Enabling Middleware (TEM) is based on the OCµ middleware and enhances it with trust capabilities. Before the TEM enhancements are explained, we give a short overview of OCµ.

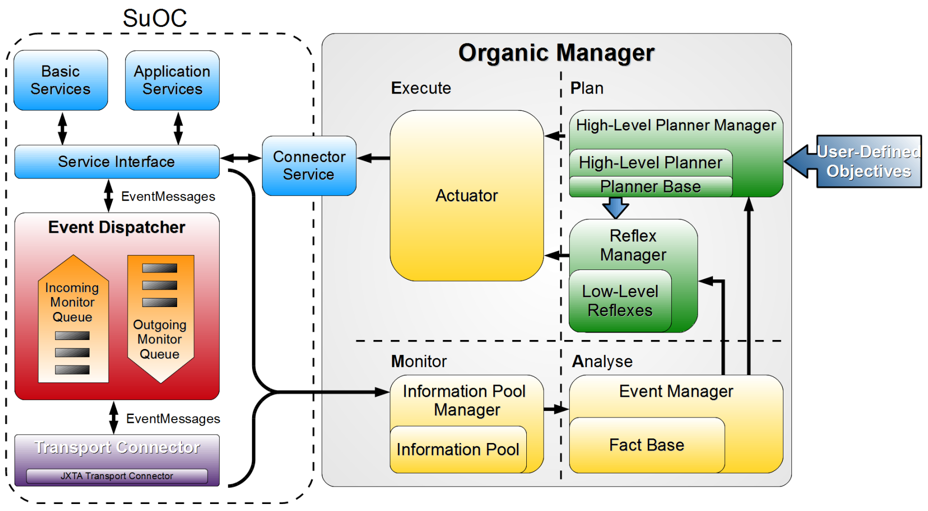

Figure 1: Architecture of OCµ

Figure 1 shows the architecture of OCµ. The left part depicts a typical middleware where services can send and receive messages. The right part, the Organic Manager, implements the self-x properties that make the middleware organic. The self-x-properties include self-configuration and self-optimization to balance the workload of nodes by relocating services. Self-healing is applied in case of node failures. With these properties, OCµ is able to autonomously react to different situations, like the failure of a node or an increase of the load of a node. The Organic Manager observes the system and uses a planner to find actions to improve the system, which are then executed by the actuator.

Services are registered and run on nodes. A TEM node is similar to a node in a Peer-to-Peer network composed of physical devices, such as PCs. Every node is identified by a unique identifier (id), the *node id*. A service is defined by implementing the `Service` interface that provides methods to interact with the rest of the system and notifications about what part of the service lifecycle (see Figure 2) the service is in.

Services are also able to register message monitors. Monitors can be added for incoming and for outgoing messages. All messages are sent through a monitor queue before being transferred to other nodes (outgoing monitor queue) or being dispatched to services (incoming monitor queue). A monitor needs to implement the `Monitor` interface and can be registered using the `MonitorManager` interface obtained by the `ServiceConnector`. It is able to add piggy-back[1] information to a message or generally observe the message flow of a node.

[1]Piggy-backing information on a message means to add additional information on top of an already existing message.

To receive messages, a service has to bind itself to the middleware. A binding is a unique string that serves as an address for other services to contact it. A binding consists of subbindings separated by a dot. All messages directed to the services registered bindings are dispatched to it. This also includes messages that contain a registered binding as subbinding. For example, if a service is registered for the binding *de.octrust.service.md5*, it receives messages with the target binding *de.octrust.service.md5*, but also messages with the binding ***de.octrust.service.md5****.encryptionRequest*. To bind itself to a binding, the service needs to call methods on a `ServiceBinder`, also obtained by the `ServiceConnector`.

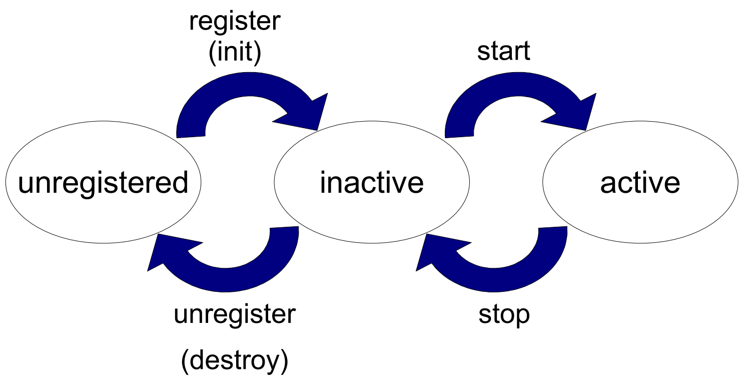

Figure 2: Service Lifecycle

Figure 2 shows the lifecycle of a service. The lifecycle consists of the following three states:

unregistered In this state, the service object was just created and can be registered on the middleware. The service can also be relocated to another node in this state. When the service is registered on the middleware, its `init` method is called and after completion of the method the service is in the inactive state.

inactive In this state, the service is registered and has a *service id* assigned to it, but does not yet receive messages. Unregistering it moves it back into the unattached state after calling the destroy method of the service. Starting the service moves it to the active state after a call to its `start` method.

active This is the state in which the service can receive messages until it is stopped again, marked by a call to its `stop` method.

The methods *init, start, stop and destroy* acknowledge a state change to a service and are described in more detail below:

init In the init phase, a service is registered with OCμ. All information to interact with the system is provided in this phase by the `ServiceConnector` interface. A service should set its binding here.

start After a service is started, it receives messages from other nodes. In this phase, monitors should be registered with the middleware, if required, by using the `MonitorManager` interface. The interface can be obtained from the `ServiceConnector`.

stop After a service is stopped, it can no longer receive messages. Messages are stored on the node until the service is started again. This phase is useful to unregister all monitors until the service is started again.

destroy In the destroy phase, the service is unregistered from OCμ and can no longer be started. In this phase, all open resources, e.g., database connections or file streams, should be closed.

If a message is dispatched to the service, a method `processMessage` of the service is called and the message is given as argument. A message in OCμ is always an `EventMessage` object. It contains the sender and the receiver as well as the payload. The sender and receiver are both identified by their node id and service binding. The binding of the sender is a reply binding where answers should be sent to. A message is created by the `EventMessageFactory` and sent by using the `MessageSender`, both obtained from the `ServiceConnector`.

To send a message, the receiving node (`targetNodeId`) and the receiving service (`destinationServiceBinding`) have to be set and the data to be sent can be added (`putElement`). If no `targetNodeId` is given, the message is broadcast to all nodes in the network. If no `destinationServiceBinding` is given, the message is broadcast to all services on the receiving nodes. Both can be combined, so a message without `targetNodeId` and `destinationServiceBinding` is broadcast to all services on all nodes. In addition, a `replyBinding` has to be set. This reply binding is the second part of the return address, besides the *node id* of the sender ((`sourceNodeId`).

The mode of the message can be set to `REQUEST`, `RESPONSE`, `EVENT`, or `FAILURE`.The modes are not interpreted by OCμ, but provide the services a way to mark the type of a message. If the message is of type `FAILURE`, it indicates a fault in the service call, and an error report, e.g., an exception, has to be added to the fault message with a specific key. `REQUEST` and `RESPONSE` are used for typical request / response communications, like the call to a md5-hasher. `EVENT` messages are typically used for one-way messages, like heartbeat-messages.

Additionally, OCμ provides a special service for proactive behavior, the `PeriodicService`, as required, for instance, in multi-agent systems. This service does not only react on received messages but also triggers a `step` method periodically. The interval can be set in each `PeriodicService` by overriding the `getInterval` method. It returns the amount of milliseconds for the intervall. Every n milliseconds, n given by the `getInterval` method, the `step` method is called. This only happens when the service is in the active state. The timer is automatically suspended in the inactive state.

As a special feature, OCμ provides a `FailureDetector` service to monitor other nodes and discover node failures [6]. The `FailureDetector` service is based on a typical heartbeat monitor, which periodically expects heartbeat messages to assure the node is still alive. The `FailureDetector` service in OCμ uses the capabilities of the organic middleware and its lazy implementation [6] adds piggy-backs on application messages by using message monitors. This reduces the message overhead created by typical failure detectors. Other services can register themselves to receive announcements of node failures.

To discover other nodes, OCµ provides a `DiscoveryService`. This service performs an asynchronous look-up to find services registered to a binding. This includes all bindings that contain the search binding as a subbinding. After a timeout given by the calling service, all services found are returned with full contact information, using the `ServiceAdvertisment` class, to the `DiscoveryListener` interface provided by the calling service.

The full Javadoc for OCµ can be found at `https://ginkgo.informatik.uni-augsburg.de/oc-trust/ocmu/javadoc/index.html`. The documents at this address will be kept up-to-date with new developments and might therefore differ slightly from the description made in this report.

The Trust-Enabling Middleware (TEM) extends OCµ with trust capabilities. A core part of this extension is the *Trust Metric Infrastructure* explained in the next section.

3 The Trust Metric Infrastructure

The objective of the Trust-Enabling Middleware (TEM) is to collect trust values and provide trust information to the application level as exemplified by the Trusted Energy Grid (TEG) in Section 4. The TEM provides techniques to handle trust operations for different contexts and is usable for measurements for different trust facets. It has to be noted that the TEM does only provide the ability to handle those trust values for facets that can be measured at run time. Both trust values from direct observation and reputation are considered in this report. Trust from direct observation is gathered from direct interactions with a communication partner. Reputation on the other hand is a trust value aggregated from information of third parties. If a potential reputation partner is not known, others in the system provide their direct trust values and a reputation trust value can be derived from this information. The nodes with direct trust values of a node that provide a reputation value from are called *neighbor* of the target node.

Trust is also highly context specific, but often, context is a role the node takes. A role is a subsumption of operations that are executed as part of an interaction by one of the interaction partners. To rate the success of an interaction, additional semantic knowledge is required. If, for example, an agent requests a MD5-hash for a string from a hashing service, the requesting agent can only determine the correctness of the result by checking the returned hash with an algorithm that can verify it. The hashing service's credibility can then be rated in its role as a "md5-hash provider". This role includes operations to accept the request, calculate the hash and return it to the requesting agent. A role is identified by a unique name. The TEM provides a specific implementation of this context, the `Role` class.

Our infrastructure provides an easy and standardized way to save trust related information and calculate trust data from this information. The infrastructure presented here also provides methods for users to calculate trust data using rather simple trust metrics as well as more sophisticated metrics. Therefore, we define the processing path as shown in Figure 3.

Services gather *raw data* about their interaction partners. These raw data represent experiences that were made with the interaction partners, e.g., measurements, and are used as a basis for the following calculations. Raw data are

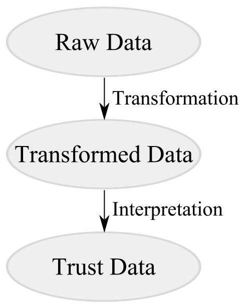

Figure 3: Multi-Stage Process: Retrieving Trust Data

then *transformed* into a preprocessed form to process them easily with an *interpreter*. Sometimes, e.g., only the most current measurements are of interest (sliding time window). Finally, these *transformed data* are interpreted into *trust data*. Trust data can either be a simple float value or a more complex object, e.g., containing time series analysis data. Figure 4 shows the UML diagram describing the Trust Metric Infrastructure, which is one of the main features of the Trust-Enabling Middleware (TEM).

The data gathered is different for each application, so the five main concepts (`RawData`, `Transformer`, `TransformedData`, `Interpreter`, `TrustData`) need to be implemented by the applications. To make things easier for standard cases, we provide default implementations for `Transformer` and `TrustData`. One of the transformers we provide is the `IdentityTransformer` that returns the given raw data. In this case, no `TransformedData` needs to be defined, instead the `RawData` implements the interface `RawTransformedData`. This class is a `RawData` as well as a `TransformedData`. We also provide a `TrustData` implementation that only consists of a single float value, the `SingleFloatTrustData`.

Services can call methods to save and get raw data, store their transformers and interpreters, and to start trust calculations. These methods are provided to the services from the TEM by an interface called `Trust`.

3.1 RawData

Raw data are the basic data gathered by the services about their interaction partners. For example, the middleware gathers information about the availability of other nodes in the form of `RawData` objects by observing message losses in order to estimate their reliability. Services as well as the middleware need to provide an implementation of the `RawData` class that contains all gathered raw data. The method `addRawData` in the abstract `RawData` class needs to be overridden and has to provide a merging logic to merge two `RawData` classes. The service can use the `Trust` interface to save new data by using the `addRawData` method. Raw data are saved per node id, facet, and context.

3.2 Transformer and TransformedData

The transformer transforms raw data into transformed data. `Transformer` is an abstract class with two methods to override: `getInputClass` and `transform`. The method `getInputClass` returns the class object of the expected `RawData`

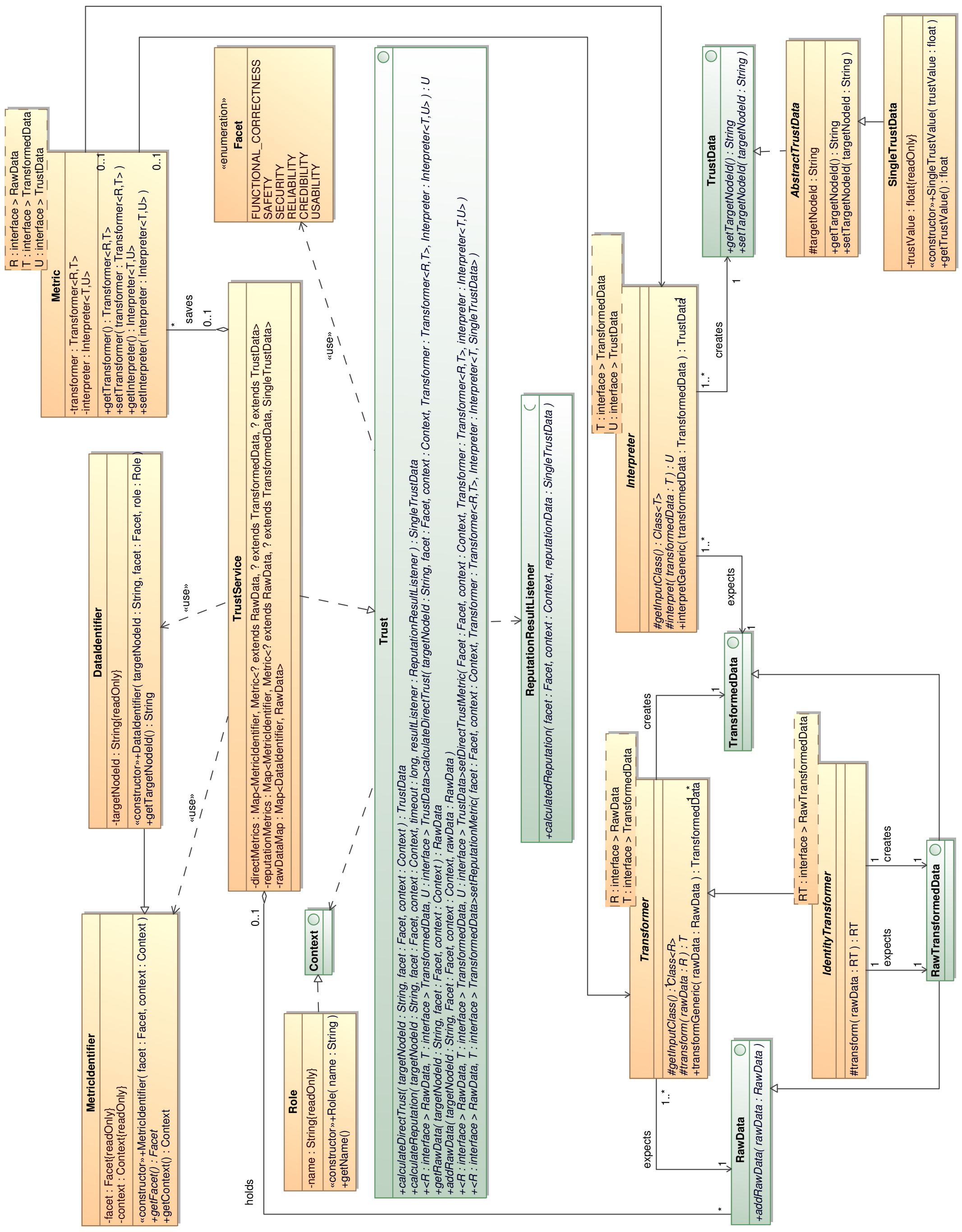
Metric
R : interface > RawData
T : interface > TransformedData
U : interface > TrustData
-transformer : Transformer<R,T>
-interpreter : Interpreter<T,U>
+getTransformer() : Transformer<R,T>
+setTransformer(transformer : Transformer<R,T>)
+getInterpreter() : Interpreter<T,U>
+setInterpreter(interpreter : Interpreter<T,U>)
«enumeration» Facet
FUNCTIONAL_CORRECTNESS
SAFETY
SECURITY
RELIABILITY
CREDIBILITY
USABILITY
Trust
+calculateDirectTrust(targetNodeId : String, facet : Facet, context : Context) : TrustData
+calculateReputation(targetNodeId : String, facet : Facet, context : Context, timeout : long, resultListener : ReputationResultListener) : SingleTrustData
+<R : interface > RawData, T : interface > TransformedData, U : interface > TrustData>calculateDirectTrust(targetNodeId : String, facet : Facet, context : Context, transformer : Transformer<R,T>, interpreter : Interpreter<T,U>) : U
+getRawData(targetNodeId : String, facet : Facet, context : Context) : RawData
+addRawData(targetNodeId : String, Facet : Facet, context : Context, rawData : RawData)
+<R : interface > RawData, T : interface > TransformedData, U : interface > TrustData>setDirectTrustMetric(facet : Facet, context : Context, transformer : Transformer<R,T>, interpreter : Interpreter<T,U>)
+<R : interface > RawData, T : interface > TransformedData>setReputationMetric(facet : Facet, context : Context, transformer : Transformer<R,T>, interpreter : Interpreter<T, SingleTrustData>)
TrustService
-directMetrics : Map<MetricIdentifier, Metric<? extends RawData, ? extends TransformedData, ? extends TrustData>
-reputationMetrics : Map<MetricIdentifier, Metric<? extends RawData, ? extends TransformedData, SingleTrustData>
-rawDataMap : Map<DataIdentifier, RawData>
DataIdentifier
-targetNodeId : String{readOnly}
«constructor»+DataIdentifier(targetNodeId : String, facet : Facet, role : Role
+getTargetNodeId() : String
MetricIdentifier
-facet : Facet{readOnly}
-context : Context{readOnly}
«constructor»+MetricIdentifier(facet : Facet, context : Context)
+getFacet() : Facet
+getContext() : Context
Role
-name : String{readOnly}
«constructor»+Role(name : String)
+getName()
Context
ReputationResultListener
+calculatedReputation(facet : Facet, context : Context, reputationData : SingleTrustData)
Interpreter
T : interface > TransformedData
U : interface > TrustData
#getInputClass() : Class<T>
#interpret(transformedData : T) : U
+interpretGeneric(transformedData : TransformedData) : TrustData
Transformer
R : interface > RawData
T : interface > TransformedData
#getInputClass() : Class<R>
#transform(rawData : R) : T
+transformGeneric(rawData : RawData) : TransformedData
IdentityTransformer
RT : interface > RawTransformedData
#transform(rawData : RT) : RT
TrustData
+getTargetNodeId() : String
+setTargetNodeId(targetNodeId : String)
AbstractTrustData
#targetNodeId : String
+getTargetNodeId() : String
+setTargetNodeId(targetNodeId : String)
SingleTrustData
-trustValue : float{readOnly}
«constructor»+SingleTrustValue(trustValue : float)
+getTrustValue() : float
TransformedData
RawTransformedData
RawData
+addRawData(rawData : RawData)
saves
holds
«use»
creates
expects
*
0..1
1
1..*

Figure 4: Trust Metric Infrastructure: Class Diagram

class. This method is needed due to the way generics are implemented in Java. The important method is the `transform` method: it expects a `RawData` class and transforms it into `TransformedData`. This `TransformedData` contains the raw data in prepared form. `TransformedData` also needs to be implemented by the application. If raw data do not need to be transformed in a specific way, the `IdentityTransformer` can be used instead. It takes and returns an object of type `RawTransformedData`, which is a `RawData` and a `TransformedData`. Because of the way generics are implemented in Java, the `getInputClass` method still needs to be overridden, but that can be done in an anonymous inner class. The `transformGeneric` method is used internally and is of no concern for applications.

3.3 Interpreter and TrustData

The transformed data can then be interpreted to yield trust data. The abstract class `Interpreter` is similar to the `Transformer`. The `getInputClass` method returns the class of the `TransformedData` that is expected as input. The `interpretGeneric` method is used internally and can therefore be ignored. The `interpret` method contains the logic to create an object of class `TrustData`, which usually represents a single float value. We provide a default implementation of this trust data, the `SingleFloatTrustData`, for such cases. If other implementations are needed, they have to be provided by the application.

3.4 Trust Interface

The Trust Metric Infrastructure can be accessed by using the `Trust` interface. This interface can be accessed through the `ServiceConnector` of OCµ. In the TEM, the `ServiceConnector` is extended to a `TrustServiceConnector`. The `ServiceConnector` can be downcast to the `TrustServiceConnector` when running a service in the TEM. The `Trust` interface provides methods to access the Trust Metric Infrastructure, to save and get raw data, and to trigger trust calculations. In the following, the methods are discussed in more detail.

addRawData, getRawData With `addRawData`, the raw data gathered by the applications can be saved into the TEM. Raw data are saved per context, facet, and node id. If a `RawData` object for such a combination already exists, the `addRawData` method inside the `RawData` object is called, where the new data is merged into the existing raw data. The method `getRawData` accordingly returns the currently saved raw data.

setDirectTrustMetric This method sets a default transformer and interpreter for a facet and context combination. The `Interpreter` must expect the `TransformedData` type returned by the `Transformer`. The input type of the transformer and the export type of the interpreter can be chosen freely. If a default metric is set, the simpler version of `calculateDirectTrust` can be called (see below).

calculateDirectTrust This method comes in two variants. Both variants expect the context, facet, and node id of the node the trust data should be calculated for. In the simple variant, no other arguments are needed and the transformer and interpreter are used, which are set by the `setDirectTrustMetric` method. The more complex version additionally expects the `Transformer` and `Interpreter` to be used. This version of the method can be used if an unusual calculation is needed for a specific facet and context combination. In most cases, the simple version that uses the default metric should suffice.

setReputationMetric This method sets the `Transformer` and `Interpreter` that are used for reputation calculations. Like the direct trust metrics, they are saved per context and facet. They are used to calculate direct trust data of a neighbor node if a request for reputation data is received. The result must always be a `SingleFloatTrustData`, therefore the `TrustData` type returned by the `Interpreter` is pre-defined. As with the direct trust metric, the `TransformedData` returned by the `Transformer` needs to be the input for the `Interpreter`.

calculateReputation This method calculates the reputation of a node. For this purpose, direct trust values from all neighbors of the target node are gathered and locally combined to an overall value. The neighbors use the `Transformer` and `Interpreter` set by the `setReputationMetric` method to calculate a direct trust value of the target node. These direct trust values are the basis for the reputation calculation on the requesting node and are therefore sent back to it. Since the communication with other nodes is asynchronous, this method does not return anything immediately. Instead, a timeout and a `ReputationResultListener` have to be provided to the method. After the timeout is elapsed, the currently received data is calculated to yield a trust value and the result is handed to the listener.

All metrics for evaluating reputation as well as direct trust, are best set in the init phase of a service.

The TEM already provides a method to gather data about the reliability of other nodes. The *Delayed-Ack-Algorithm* [3] uses the monitoring ability of the TEM to add piggy-back information on messages to observe the message flow of application messages. By doing so, the algorithm can determine the targets of application messages and whether or not the application messages were received. From this data, it can calculate a reliability value per node and save it data into the Trust Metric Infrastructure. The reliability data can be read by calling the calculate methods with a special context and the facet reliability. The Delayed-Ack-Algorithm therefore provides a `Transformer` and `Interpreter` for the raw data it gathers.

The classes at the top of the diagram, `MetricIdentifier`, `DataIdentifier`, `Metric`, and `TrustService` are internal classes to implement the logic of the infrastructure and are not further explained.

The full Javadoc of the TEM can be found at `https://ginkgo.informatik.uni-augsburg.de/oc-trust/tem/javadoc/index.html`. The documents provided at this address will be kept up-to-date with new developments and might therefore differ slightly from the description made in this report.

In the next section, an application of the TEM and its Trust Metric Infrastructure to a system that deals with uncertainty in power grids is shown.

4 Application of the Trust-Enabling Middleware

The *Trusted Energy Grid (TEG)* is a platform providing an infrastructure for applications based on a power grid with producers and consumers. It uses the Trust-Enabling Middleware (TEM) as underlying middleware framework and makes use of the functionality and interfaces the TEM provides, including the Trust Metric Infrastructure. Due to the number of producers and consumers, the fact that the power grid always has to work properly, and the potential damage in case of failures, the Trusted Energy Grid is a large-scale, safety-critical system.

In this section, a scenario in which the Trusted Energy Grid allows an application to deal with these problems is described. The concepts of the Trusted Energy Grid and the application are then instantiated based on the abstractions provided by the Trust-Enabling Middleware.

4.1 Scenario

The big challenge in power networks is to balance energy production and consumption while considering uncertainty introduced by stochastic energy sources and by a fluctuating energy demand. Stochastic power sources are, for example, energy sources based on natural resources like wind and sun, or domestic combined heat and power (CHP) units. Both have in common that they are less controllable than regular power sources and are rather unpredictable regarding their power output, because they either depend on weather conditions or the customer's behavior. Moreover, especially the tremendously increasing number of stochastic power sources exacerbates the situation. The system's task is to master this situation and to plan and operate under these conditions while maintaining safety.

An application on top of the Trusted Energy Grid are *Autonomous Virtual Power Plants (AVPPs)* [1], which are an approach to hold the balance between energy consumption and energy production despite the rising complexity and uncertainty introduced by stochastic power sources and by a fluctuating energy demand. For this purpose, the power plant landscape is partitioned into several clusters, the AVPPs. An AVPP is a self-organizing, self-adapting, and self-optimizing ensemble of power plants. It autonomously plans the energy supply based on predictions made by producers and consumers, and it reacts to load or supply changes by adjusting schedules. The composition of an AVPP is not fixed; on the contrary, AVPPs change their structure from time to time if necessary, for example, when the AVPP permanently cannot cope with the load.

AVPPs use trust [7] as a way to handle uncertainty. Each power plant's trustworthiness is characterized by two aspects, credibility and reliability. The credibility of a power plant is understood as the accuracy of its output predictions. Hence, stochastic power plants are most likely less credible than baseload plants. A power plant's reliability describes how often it had technical problems and, therefore, was off-line or had to reduce its power output unexpectedly.

On the one hand, trustworthiness of power plants is used to plan the energy supply. Untrustworthy power plants are less involved in supplying energy than trustworthy power plants in order to reduce and limit possible imbalances between energy production and consumption that could be caused by failing power plants or unsteady power outputs. On the other hand, trustworthiness of power plants is considered during the formation of AVPPs. In order to make the system more robust, AVPPs ought to be of similar quality which all can cope equally well with their tasks. Hence, an AVPP mainly consisting of untrustworthy power plants – that cannot cope with its task – is not desired, and the formation mechanism of AVPPs thus tries to achieve a good mix of trustworthy and untrustworthy power plants for every AVPP.

In order to enable AVPPs to use trust, experiences with power plants have to be measured, collected, and interpreted to yield trust data. Thus, the TEM's Trust Metric Infrastructure suits the needs of AVPPs well. In the following, the mapping of the TEM and its Trust Metric Infrastructure to the Trusted Energy Grid and AVPPs is described.

4.2 Usage of the Trust-Enabling Middleware

Each power plant in the Trusted Energy Grid is a periodic service of the TEM, thus achieving a proactive behavior that is needed for the power plants. Moreover, by being a periodic service, a power plant inherits several other required capabilities, such as the ability to communicate with other services (which is needed, e.g., in the process of forming AVPPs), to find other services, and to start or stop itself when it goes on- or off-line.

The Trusted Energy Grid makes use of the TEM's built-in feature to collect reliability raw data of a node, i.e., the node's availability. Because a node in the middleware represents a physical device, each power plants runs on its own node.

Since credibility depends on the application, credibility raw data cannot be collected by the TEM itself. Thus, this has to be done by the application. Therefore, the Trusted Energy Grid uses the TEM's feature to store credibility raw data in the TEM itself. As presented in Sect. 3, the Trust Metric Infrastructure provides the possibility to enhance the TEM by application-specific trust metrics. Since deriving credibility values for power plants is a highly complicated and application-specific process, specialized trust metrics using the Trust Metric Infrastructure are needed. In the following, a trust metric for deriving credibility values is outlined.

Raw data in terms of a power plant's credibility is a list of predicted and actual power outputs of the power plant that is stored in the TEM. This list can then be transformed by a special *transformer* to *transformed data* which are, for example, the average percental difference between actual output and predicted output of all raw data, and/or some statistical data like the standard deviation. The transformed data are interpreted by a special *interpreter*, considering, e.g., the positive or negative difference between actual and predicted output. A positive difference, that is, the power plant produces in average more power than predicted, might be regarded less critical than a negative difference. This is due to the fact that a positive difference could act as a little cushion, and, if not needed, be sold to other AVPPs. The time of day at which the prediction was made is of special interest within the application of AVPPs as credibility at peak load times like during midday is more important than credibility at night. For that reason, the time of day is represented by the *context*. This results in *trust data* for different time slots, for example, a credibility value for morning, a credibility value for midday, and so on.

Even though the Trust-Enabling Middleware not only provides an infrastructure for deriving trust through direct observation but also provides the possibility to use reputation, this feature is currently not used in the application of AVPPs. Instead, it is assumed that power plants honestly store their respective credibility raw data themselves and reliability raw data are collected by the TEM, and that everyone can directly access these raw data or the resulting trust data so that no reputation is needed.

The usage of the Trust-Enabling Middleware in the Trusted Energy Grid has been described rather abstractly so far and is detailed in an actual instantiation of parts of the Trusted Energy Grid in the next section.

4.3 Instantiation Example

In Figure 5, a simplified class diagram of the instantiation of the Trust Metric Infrastructure in the Trusted Energy Grid (TEG) is shown. It only illustrates the classes for the trust facet credibility; the classes for the facet reliability look similar. Classes beginning with "TEG" depict the application-specific classes of the Trusted Energy Grid.

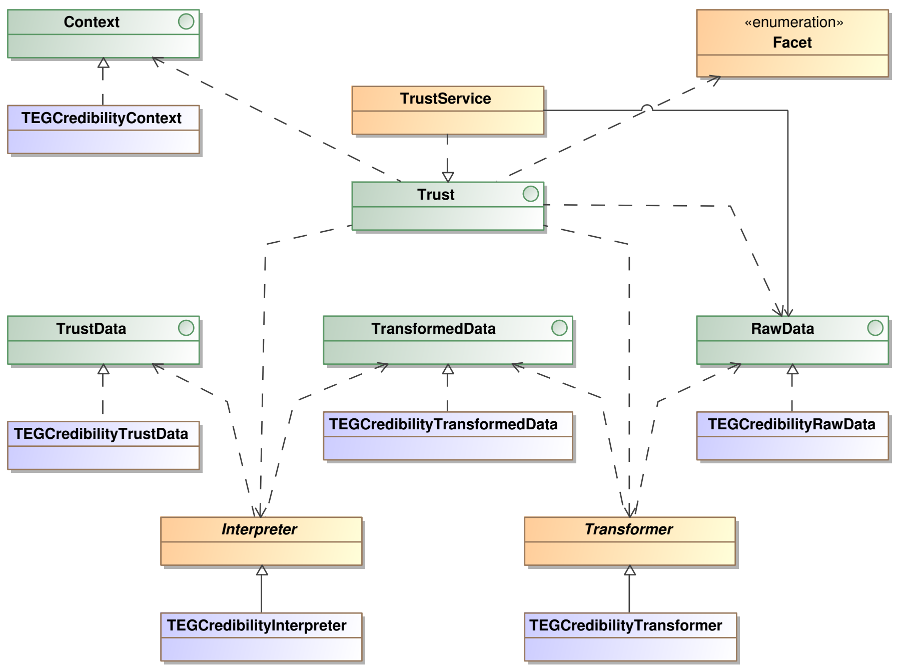

Figure 5: Application of the Trust Metric Infrastructure in the Trusted Energy Grid (TEG): Simplified Class Diagram

The following outline briefly describes how to instantiate and use the Trust Metric Infrastructure.

1. By the use of an implementation of the TEM's interface `Trust` – the TEM's provided `TrustService` – each power plant gets the capability of

adding raw data and calculating trust data with the help of a transformer and an interpreter.

2. For this purpose, raw data, transformed data, and trust data have to be specified in the Trusted Energy Grid. This is done by implementing the TEM's interfaces `RawData`, `TransformedData`, and `TrustData`, resulting in `TEGCredibilityRawData`, `TEGCredibilityTransformedData`, and `TEGCredibilityTrustData`.

3. Moreover, in the Trusted Energy Grid, the abstract classes `Transformer` and `Interpreter` of the TEM have to be extended to know how to transform `TEGCredibilityRawData` to `TEGCredibilityTransformedData` and how to interpret these to yield `TEGCredibilityTrustData`. Thus, the classes `TEGCredibilityTransformer` and `TEGCredibilityInterpreter` are obtained.

4. The `TEGCredibilityTransformer` and the `TEGCredibilityInterpreter` have to be made known to the `Trust` interface by calling the method `setDirectTrustMetric`.

5. If `TEGCredibilityRawData` are measured, they can be stored in the TEM by calling `addRawData` provided by the interface `Trust`.

6. Whenever `TEGCredibilityTrustData` are needed, the interface `Trust` provides the method `calculateDirectTrust`. This method automatically uses the defined transformer and interpreter. However, the interface `Trust` also provides an overloaded version of the method `calculateDirectTrust` in which another transformer and interpreter can be passed (see Sect. 3).

5 Conclusion and Future Work

In this report, we presented the Trust-Enabling Middleware (TEM) that is based on the organic middleware OCµ. Since services in OCµ only act reactively, the TEM introduces the concept of periodic services, thus obtaining a proactive behavior. Services can communicate with other services by exchanging messages. This makes it possible for the TEM to act as the basis for multi-agent systems. Additionally, the TEM provides a failure detector which monitors nodes and thus observes if a node goes off-line.

Furthermore, the Trust-Enabling Middleware provides the Trust Metric Infrastructure for storing, interpreting, and querying trust data for a specific context and facet of trust. For this purpose, experiences that were made with interaction partners in a specific context and for a specific facet are stored as raw data and then transformed to transformed data. These are interpreted to yield trust data afterwards. The Trust-Enabling Middleware has a built-in mechanism to collect reliability raw data by monitoring message losses. Further, it already provides basic metrics. However, it is possible to define application-specific, more sophisticated trust metrics by developing customized transformers and interpreters, especially for trust facets like credibility which cannot be interpreted by the middleware itself as credibility is a highly application-specific property.

Finally, we showed an example application, the Trusted Energy Grid, that is based on the Trust-Enabling Middleware and makes use of the features the middleware offers. In particular, we showed what a complex trust metric looks like in the Trusted Energy Grid, and illustrated how to implement it using the interfaces of the TEM. Another example application that uses the TEM as underlying middleware is the Trusted Computing Grid [2], a desktop grid computing system that uses trust as a way to increase performance and robustness.

Future work includes the fine-tuning of the middleware's architecture, the Trust Metric Infrastructure, and the interfaces. Moreover, the Trust-Enabling Middleware will be extended by further features. For instance, the self-x properties of the middleware will be improved in such a way that they consider the trustworthiness of nodes. So it will be possible that the TEM's ability to self-optimize migrates services to other nodes (load balancing) while considering the trustworthiness of nodes: it is better to migrate a service to a trustworthy node than to migrate it to an untrustworthy node. Further, metrics aggregating direct trust values and reputation will be developed. A node's confidence, i.e., how much a node trusts its own experiences, will play an important role in this case. Last but not least, a persistent storage will be included in the TEM. This will be used by the services which then do not have to take care of a persistent storage of data themselves.

Acknowledgment

This research is partly sponsored by the research unit "OC-Trust" (FOR 1085) of the German Research Foundation (DFG).

References

[1] G. Anders, F. Siefert, J.-P. Steghöfer, H. Seebach, F. Nafz, and W. Reif. Structuring and Controlling Distributed Power Sources by Autonomous Virtual Power Plants. In *Proceedings of the Power & Energy Student Summit 2010 (PESS 2010))*, pages 40–42, October 2010.

[2] Y. Bernard, L. Klejnowski, J. Hähner, and C. Müller-Schloer. Towards Trust in Desktop Grid Systems. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 637–642, Los Alamitos, CA, 2010. IEEE Computer Society.

[3] R. Kiefhaber, B. Satzger, J. Schmitt, M. Roth, and T. Ungerer. Trust Measurement Methods in Organic Computing Systems by Direct Observation. In *Proceedings of the 8th International Conference on Embedded and Ubiquitous Computing (EUC 2010)*, pages 105–111. IEEE, 2010.

[4] C. Müller-Schloer. Organic Computing – On the Feasibility of Controlled Emergence. In *CODES+ISSS '04: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 2–5, Washington, DC, USA, 2004. IEEE Computer Society.

[5] S. Ramchurn, D. Huynh, and N. Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(01):1–25, 2005.

[6] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer. A Lazy Monitoring Approach for Heartbeat-Style Failure Detectors. In *Proceedings of the The Third International Conference on Availability, Reliability and Security (ARES 2008), Technical University of Catalonia, Barcelona, Spain*, pages 404–409, March 4-7 2008.

[7] J.-P. Steghöfer, R. Kiefhaber, K. Leichtenstern, Y. Bernard, L. Klejnowski, W. Reif, T. Ungerer, E. André, J. Hähner, and C. Müller-Schloer. Trustworthy Organic Computing Systems: Challenges and Perspectives. In *Proceedings of the 7th International Conference on Autonomic and Trusted Computing (ATC 2010)*. Springer, October 2010.

[8] W. Trumler. *Organic Ubiquitous Middleware*. PhD thesis, University of Augsburg, 2006.