Universität
Augsburg
University

TEMAS – A Trust-Enabling Multi-Agent System for Open Environments

Gerrit Anders, Florian Siefert, Nizar Msadek, Rolf Kiefhaber, Oliver Kosak, Wolfgang Reif, Theo Ungerer

**Institut für Informatik
Universität Augsburg**

Report 2013-04 April 2013

Institut für Informatik
D-86135 Augsburg

TEMAS –
A Trust-Enabling Multi-Agent System for Open Environments

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

Institute of Computer Science

Augsburg University, Germany

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

Oliver Kosak

Student at the Institute of Computer Science

Augsburg University, Germany

E-Mail: oliverkosak@googlemail.com

The TEMAS – the Trust-Enabling Multi-Agent System – is a multi-agent system for open environments. It is based on the Trust-Enabling Middleware, which itself is based on the adaptive, organic middleware OCµ that features self-x properties such as self-healing and self-optimization. Further, the TEMAS incorporates an infrastructure that provides a variety of multi-agent system concepts. Apart from facilities for communication in local and distributed environments and a yellow pages service, it allows itself and the agents to use application-specific metrics to derive trust values for different facets from prior experiences with the Trust Metric Infrastructure provided by the Trust-Enabling Middleware. In the TEMAS, agents can be run on nodes, a form of container similar to those used in peer-to-peer networks. Nodes often represent physical devices and can host several agents or reactive services. With respect to the Trust-Enabling Middleware, the TEMAS serves as a facade because it hides the complexity of the underlying infrastructure consisting of nodes and services and dependent interfaces to higher level applications. This results, e.g., in simpler, more common, and natural interfaces for messaging and the application of trust in multi-agent systems.

Contents

1 The Trust-Enabling Multi-Agent System

Participants in open, heterogeneous multi-agent systems (MAS) have to deal with numerous uncertainties. These arise from the following properties: In such systems, agents (1) are embedded in a dynamic, potentially hostile environment, (2) only have very limited knowledge about the behavior of other agents as well as the underlying infrastructure, and (3) only have very limited control over these agents and their environment. Consequently, agents might be confronted with interaction partners that do not behave as expected or desired, and there even might be agents that try to cheat or damage the system. Further, the MAS itself must be able to deal with uncertainties that arise due to unreliable computational devices constituting the system's infrastructure such as unreliable PCs or smartphones. It is therefore crucial that agents as well as the underlying infrastructure are able to identify interaction partners and devices they can rely on to fulfill the agents' own or the system's goals.

Trust is a multi-faceted concept that allows system participants to gauge and cope with these uncertainties. It includes, among others, credibility and reliability [14]. Credibility specifies an agent's willingness to participate in an interaction in a desirable manner, and corresponds to the original notion of trust in MAS [10]. Reliability indicates its quality of an agent or a device with regard to its availability under disturbances or partial failure.

MAS that are deployed in open environments must be able to provide mechanisms that allow both the underlying infrastructure as well as the agents to make use the concept of trust to measure uncertainties at runtime and to take the trustworthiness of agents and devices into account when making decisions.

On the one hand, an agent that makes decisions on the basis of the trustworthiness of potential interaction partners (e.g., when searching for a suitable contractor at an electronic market) can increase its benefit because it is likely that an interaction partner with a high trust value behaves more beneficial than an agent with a low trust value. Therefore, the risk to interact with less trustworthy agents might be higher. On the other hand, a trust-aware MAS can ensure that important services and agents are hosted on reliable devices in order to increase the overall system's stability and availability.

The *Trust-Enabling Multi-Agent System* (TEMAS) is a MAS for open environments. It is based on the *Trust-Enabling Middleware* (TEM) [6], which itself is based upon and extends the adaptive, organic middleware *OCμ* [11]. While OCμ features self-x properties such as self-healing and self-optimization, the TEM basically extends OCμ by the notion of trust. The TEMAS incorporates an infrastructure that allows itself and the agents to use application-specific metrics to derive trust values for different facets from prior experiences. In addition to the possibility to derive trust values, the TEMAS allows to store experiences in persistent data bases, query reputation values (i.e., trust values that originate from experiences of other system participants), and assess the reliability of agents and devices out of the box. Besides this mature trust infrastructure and mechanisms, of course, the TEMAS supports communication in local and distributed environments, defines basic agent concepts and exhibits common services such as yellow pages.

In the following, we introduce the most important concepts and basic mechanisms

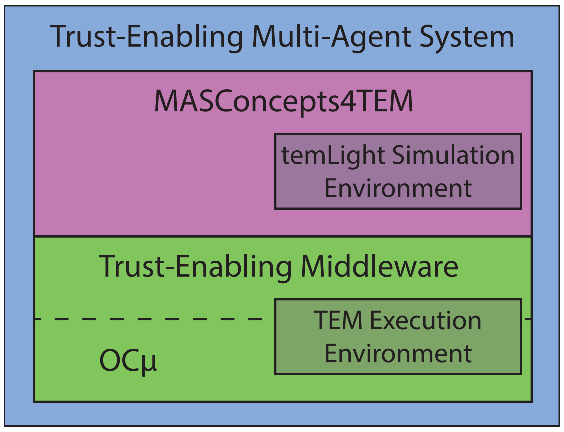
Trust-Enabling Multi-Agent System
MASConcepts4TEM
temLight Simulation Environment
Trust-Enabling Middleware
TEM Execution Environment
OCμ

Figure 1: The TEMAS architecture

of the TEMAS that are worth knowing when coming into touch with the TEMAS. We introduce these concepts and mechanisms in an order according to the TEMAS's architecture (see Figure 1). The TEMAS basically consists of two layers. The lower layer is the TEM (see Section 2), whereas the upper layer is called *MASConcepts4TEM* (see Section 3). The MASConcepts4TEM layer defines concepts which are necessary in MAS, but not already available on the middleware, i.e., TEM, layer. Furthermore, it hides the complexity of the underlying infrastructure and interfaces to higher level applications. Among others, the MASConcepts4TEM include the concept of agents. Having introduced the basic concepts, mechanisms, and their interrelations, we give advice on how to use and create projects on the basis of the TEMAS in Section 4. Finally, we conclude the technical report in Section 5.

2 The Trust-Enabling Middleware

The Trust Enabling Middleware (TEM) enhances the basic OCµ middleware that incorporates self-x properties, i.e., the ability to heal, protect, configure, and optimize itself, with trust capabilities. Integrating trust into such a system can considerably improve the effectiveness of the self-x properties. An example would be to preferably assign services to trustworthy nodes during self-configuration. Such as mechanism will reduce the need for self-healing since trustworthy nodes can be assumed to be more reliable. Therefore, trust is an important aspect in the TEM.

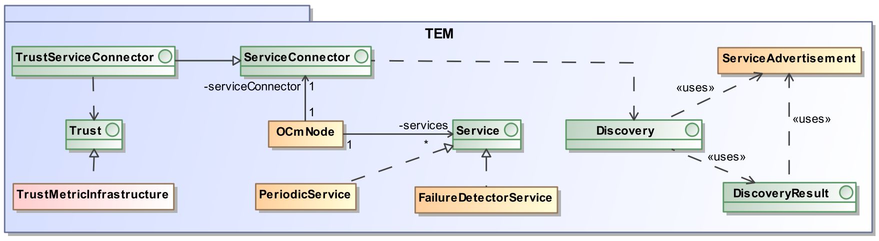

Figure 2: Concepts of the Trust-Enabling Middleware. The middleware's trust functionality is encapsulated in the concept `TrustMetricInfrastructure` which represents a variety of concepts that implement the actual functionality. A more detailed view on these concepts is provided in Section 2.7.

The most important concepts of the TEM are depicted in Figure 2. In the following, we take a closer look at these concepts. The runtime environment provided by the TEM, called *TEM execution environment*, and an overview of how systems that are based on this middleware and the TEMAS can be deployed is explained in Section 4.

2.1 OcmNode

In the TEM, services are registered and run on nodes. A node in the TEM is similar to a node in a peer-to-peer network which is composed of physical devices such as PCs. Every node is identified by a unique identifier (id). In the TEM, calling the method `buildTEMNode` of the class `TEMFactory`, extends the given `OcmNode` to a TEM node, i.e., an `OcmNode` with trust capabilities. To obtain an `OcmNode`, several factories are available:

- `OcmLocalFactory` for local nodes that can communicate within a single JVM.
- `OcmJxtaFactory` for creating nodes with the JXTA[1] `TransportConnector`.
- `OcmFailureFactory` for a wrapper around all `TransportConnectors` to inject errors in the transport layer, e.g., missing or delayed messages.

[1]**JXTA:** Open source peer-to-peer protocol specification begun by Sun Microsystems in 2001 (see http://jxta.kenai.com/)

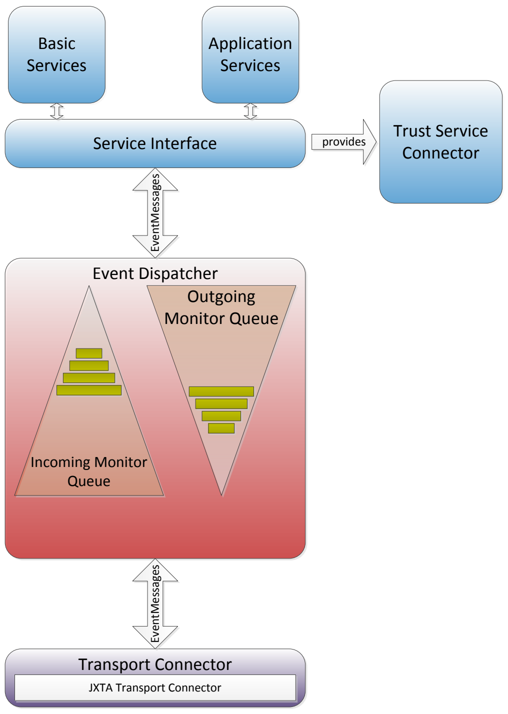

Figure 3: TEM node Architecture

Figure 3 depicts the architecture of a TEM node. It is split in three main parts:

1. **Transport Connector:** To decouple the TEM from the underlying communication infrastructure, the Transport Connector layer provides interfaces for different communication infrastructures (i.e., local or distributed communication). However, nodes by the distributed version must be able to communicate fast and reliable with other nodes. The best way to guarantee this is the use of a peer-to-peer network. In the current implementation, we use JXTA and provide a JXTA `TransportConnector`. The implementation of a Transport Connector can be replaced depending on the given communication infrastructure, which is transparent to the rest of TEM and the applications built on top of it. It is also possible to use multiple `TransportConnector` implementations for different communication infrastructures at the same time (e.g., UDP, Bluetooth, Serial Line).

2. **Services:** Services run on nodes. They implement and provide the functionality of the system. Furthermore, servies are able to register message monitors. Monitors can be added for incoming and outgoing messages. All messages are sent through a monitor queue before being sent to other nodes (outgoing monitor queue) or being dispatched to services (incoming monitor queue). Services will be discussed later in more detail in Section 2.2.

3. **Trust Service Connector:** The `ServiceConnector` of OCµ provides interfaces

that can be used by the services to interact with the middleware, e.g., to create and send messages. The `TrustServiceConnector` extends the `ServiceConnector` to provide access to the trust capabilities of the TEM through the `Trust` interface. The important methods of the `ServiceConnector` and `TrustServiceConnector` are described below:

- **ServiceConnector**
  - **getMessageSender:** Used to send messages to other services.
  - **getServiceBinder:** Used to bind services to message bindings. The bindings are explained in detail in Section 2.2.
  - **getMessageFactory:** Used to retrieve a factory to create objects of type `EventMessage`. Messaging is explained in detail in Section 2.3.
  - **getMonitorManager:** Used to register monitors for incoming and outgoing messages.
  - **getDiscovery:** Used to find other services in the network. It is described in detail in Section 2.6.
- **TrustServiceConnector**
  - **getTrust:** Used to get an interface to save experiences and start trust calculations. This interface is explained in detail in Section 2.7.

2.2 Service

In order to receive messages, a service has to register its binding with the TEM, i.e., the service is bound to this binding. A binding is a string that serves as an address for other services to contact it. Furthermore, a binding consists of subbindings separated by a dot. All messages directed to the service's registered bindings are dispatched to it. This also includes messages that contain a registered subbinding. For example, if a service is registered for the binding `de.octrust.service.md5`, it receives messages with the destination service binding `de.octrust.service.md5`, but also messages with the destination service binding `de.octrust.service.md5.encryptionRequest`. In general the service binds itself on a general binding and receives all messages that are sent to more specific bindings. To bind itself to a binding, the service needs to call methods on a `ServiceBinder`, obtained by the `TrustServiceConnector`.

A service needs to implement the `Service` interface to be able to run on the TEM. It provides methods to interact with the rest of the system and notifications of what part of the service life cycle (see Figure 4) the service is in. The life cycle consists of the following three states:

- **unregistered:** In this state, the service object was just created and can be registered with the middleware. When the service is registered, its `init` method is automatically called. After completion of the method, the service is in the state "inactive".

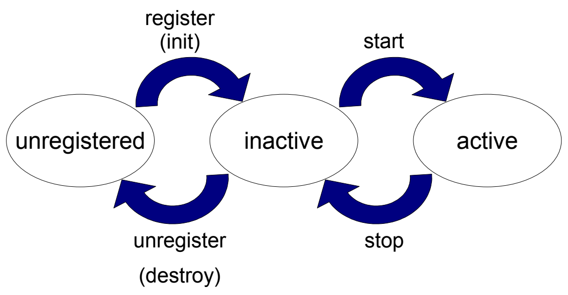

Figure 4: Life cycle of a `Service`

- **inactive:** Here, the service is registered and has a service id assigned to it, but does not yet receive messages. In this state, the service can also be relocated to another node. If it is unregistered, it is moved back into the state "unregistered" after automatically calling the `destroy` method of the service. If the service's method `start` is called, the service changes its state to "active".
- **active:** This is the state in which the service can receive messages until it is stopped again. A service can be stopped by calling its method `stop`.

The `Service` interface provides all methods mentioned above, such as `init`, `start`, and `stop`. These are described in more detail below:

```
public interface Service
```

Important Methods

- **init:** The `init` method is automatically called when the service is registered on a TEM node. Several information is provided to interact with the middleware. Here, the service should register its bindings with the `ServiceBinder`. It will automatically be bound to its own id (`serviceId`). The `ServiceConnector` contains the interfaces to interact with the middleware. In case of the TEM, it can be downcasted to `TrustServiceConnector`. The `initialData` object is the same as in the `destroy` method when the service is started after a relocate. If an error occurs, the method can throw an `InitializationException` to notify the middleware.

```
public void init(String serviceId, ServiceConnector serviceConnector,
    Map<String, Serializable> initialData)
    throws InitializationException;
```

- **start:** Having called the method `start`, the service can receive messages. In this phase, monitors can be registered with the middleware by using the interface `MonitorManager`. If an error occurs while starting the service, the middleware can be notified by throwing a `ServiceStartException`.

```
public void start() throws ServiceStartException;
```

- **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. If an error occurs, a `ServiceStopException` can be thrown.

```
public void stop() throws ServiceStopException;
```

- **destroy:** This method is called prior to the removal of the service from the node. Its bindings will be removed automatically. The service should perform cleanup operations if needed, i.e., database connections or file streams should be closed. The service can store its current state in the given map in case it was unregistered due to a relocation. The same map is then used as a parameter when calling the `init` method.

```
public void destroy(Map<String, Serializable> transferData);
```

- **processMessage:** This method is called to dispatch the given message to the service.

```
public void processMessage(EventMessage message);
```

- **getName:** Through this method the service can define a human readable name for itself. It is not called by the middleware but can be used, e.g., to display services in a GUI.

```
public String getName();
```

- **getServiceId:** After the service is initialized, this method must return the id of the service which was given as an argument when the `init` method was called. The structure of the serviceId is described in more detail further below.

```
public String getServiceId();
```

- **getServiceType:** Returns the service type of this service. This typically is the fully qualified class name as specified by the command `getClass().getName()`.

```
public String getServiceType() {
    String result = getClass().getName();
    return result;
}
```

The five four mentioned methods from above, i.e., `init`, `start`, `stop`, `destroy`, and `stop` can only be triggered internally.

ServiceId The service id is a globally unique identifier of a service. It consists of two parts:

1. The service type
2. An additional id part

The service type is defined by the method `getServiceType`. The id part is generated by the middleware. Both parts are separated by the character #. The class `Util` defines some static helper methods to process service ids, e.g., to split a service id into its parts or extracting the type from an id.

Usage The following code fragment demonstrates how to create a local TEM node, i.e., a node that can communicate within a JVM:

```
//Create a local OcmNode with ascending node ids and extend
//it to a TEM node.
IdFactory idFactory = new AscendingIdFactory();
OcmNode node1 = OcmLocalFactory.createLocalNode(idFactory);
node1 = TEMFactory.buildTEMNode(node1);

//Create a new service object.
DummyService dummy1 = new DummyService();

//Register and start the newly created service on the TEM node.
node1.registerService(dummy1);
node1.startService(dummy1);
```

At first, a new instance of the class `AscendingIdFactory` is created to provide ascending ids for nodes and services.

```
IdFactory idFactory = new AscendingIdFactory();
```

The `IdFactory` is given as argument to the method `createLocalNode` to create a new instance of the class `OcmNode` (here `node1`). Alternatively, the method can be called without an argument, in that case the default `IdFactory` (`IdFactoryImpl`), which creates globally unique, yet somewhat cryptic, ids, is used.

```
OcmNode node1 = OcmLocalFactory.createLocalNode(idFactory);
```

In order to extend our generated `node1` to a TEM node with trust capabilities, the factory method `buildTEMNode` has to be called, which expects the corresponding `OcmNode` as argument.

```
node1 = TEMFactory.buildTEMNode(node1);
```

Then we create one dummy Service, called dummy1.

```
DummyService dummy1 = new DummyService();
```

By calling the method `registerService` the service will be registered with the TEM node. Then `startService` starts the service.

```
node1.registerService(dummy1);
node1.startService(dummy1);
```

2.3 Messaging

There are four different ways of addressing and transmitting a message over the TEM that are described in more detail below:

- **Unicast Message with Binding:** The message is sent to all services that are bound to the specified message binding and run on a specific node. A service uses this kind of message to send a request or response directly to another service.

- **Broadcast Message with Binding:** The message is sent to all services in the system that are bound to the specified message binding.

- **Unicast Message without Binding:** The message is sent to all services that run on a specific node. It can be used to get an alive response from all services on that node.

- **Broadcast Message without Binding:** The message is sent to all services in the system. It can be used to get an alive response from all running services.

If a message is dispatched to the service, the method `processMessage` of the service is called and the message is given as argument. A message in the TEM 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. An `EventMessage` is created by the `EventMessageFactory` and sent by using the `MessageSender`, both obtained from the `ServiceConnector`. To send an `EventMessage`, the receiving node (`destinationNodeId`) and the receiving service (`destinationServiceBinding`) have to be set. Data that is to be sent can also be added (`putElement`) to the message. If no `destinationNodeId` is given, the message is a broadcast to all nodes in the network. If no binding is given, the message is a broadcast to all services on the `destinationNodeId`. Both can be combined so that a message without information about `destinationNodeId` and `destinationServiceBinding` is a broadcast to all services on all nodes. In addition, a `replyBinding` has to be set. This `replyBinding` 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 the TEM, but provide 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 that is defined as a constant in the `EventMessage` (`KEY_FAILURE`). `REQUEST` and `RESPONSE` are used for typical request/response communications, like the call to a md5-hasher. `EventMessage`s are typically used for one-way messages, like heartbeat messages.

Usage In the following, we demonstrate how to create and send an EventMessage object from a given node1 to another given node2. First, a new unicast EventMessage is created. All required arguments are described in more detail below:

- **mode:** The mode of the EventMessage.
- **sourceServiceId:** Id of the service, this EventMessage is sent from.
- **replyBinding:** The serviceBinding the answers to this event should be sent.
- **destinationNodeId:** the id of the node this EventMessage is assigned to.
- **destinationServiceBinding:** The string the destination service is bound to on the destination node. The id of a service can be set here to mark a specific service as receiver. If null is set, all services on the node will receive this message.

```
EventMessage dummyMsg = node1
    .createUnicastMessage(
    //Mode
    Mode.REQUEST,
    //SourceServiceId
    dummyService.getServiceId(),
    //ReplyBinding
    "de.octrust.service.md5.encryptionRequest",
    //DestinationNodeId
    node2.getId(),
    //DestinationServiceBinding
    "de.octrust.service.md5.encryptionResponse");
```

In order to send the newly generated dummyMsg to node2, the sendMessage method of node1 has to be called, which expects the message to send.

```
node1.getServiceConnector().getMessageSender().sendMessage(dummyMsg);
```

2.4 PeriodicService

```
public abstract class PeriodicService implements Service
```

The abstract service PeriodicService is a particular service for proactive behavior that implements the Service interface and is used, e.g., in multi-agent systems to implement agents (see Section 3). Figure 5 depicts the life cycle of a PeriodicService.

It consists of the following states:

- **idle:** This state tells the service to give up any activity and sleep until the given time, specified in milliseconds, elapses, whereupon the service changes its state back to “active”.
- **active:** In this state, the step() method is called once and, after its execution, the state moves back to “idle”.

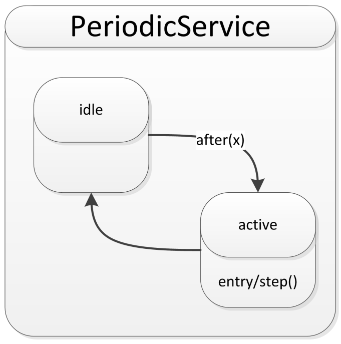

Figure 5: Life cycle of a `PeriodicService`

Important Members

- **timer:** Used to schedule the periodic calls of the `step` method.

```
private Timer timer;
```

- **exec:** `Task` for the timer to execute the `step` method in time.

```
private final TimerTask exec = new TimerTask() {
    @Override
    public void run() {
        step();
    }
};
```

Important Methods

- **start:** The `start` method is overridden in order to create the underlying thread that calls the `step` method. This is a final method and cannot be overridden in a subclass. The `startService` method, in contrast, can be overridden and is called by `start` .

```
@Override
public final void start() {
    timer = new Timer();
    timer.schedule(exec, getInterval(), getInterval());
    startService();
}
```

- **stop:** The `stop` method is overridden in order to stop the underlying thread to call the `step` method. This is a final method as well and can not be overridden in a subclass. The `stopService` method, in contrast, can be overridden and is called by the `stop` method.

```
@Override
public final void stop() {
    timer.cancel();
    stopService();
}
```

- **getInterval:** This method must return the amount of milliseconds specifying the frequency with which the underlying thread calls the `step` method. This only happens when the service is in the state "active". The timer is suspended in the state "inactive".

  ```
  protected abstract long getInterval();
  ```

- **step:** This method is called automatically when the state "active" is entered. After its execution, the state changes back to "idle".

  ```
  protected abstract void step();
  ```

2.5 FailureDetectorService

The service `FailureDetectorService` is used to monitor other nodes and discover node failures within the TEM [12]. It does also find other nodes to monitor itself. This is done by using a lazy heartbeat monitor, which periodically expects heartbeat messages of nodes to assure they are still alive, while using already existing application messages to piggy-back heartbeat messages to reduce the amount of required messages. Other services can receive announcements of node failures by binding themselves to the binding `FailureDetectorService.BINDING_FAILED`. When such a message is received, the `EventMessage` will contain data for the following keys (this data can be accessed by calling `getElement(String key)`):

- **FailureDetectorService.MESSAGE_KEY_FAILED_NODE**: Contains the id of the node that is marked as failed.

- **FailureDetectorService.MESSAGE_KEY_FAILURE_PROBABILITY**: This key contains the probability with which the node has failed, which triggered the message, a float value between 0 and 1.

The `FailureDetectorService` can be configured by using the configuration files described in Section 4.6.

2.6 Discovery

2.6.1 Discovery Interface

```
public interface Discovery
```

The `Discovery` interface can be obtained from the `ServiceConnector` interface, which is obtained by the `init` method of a `Service`. It provides the capability to search for services in the network based on their registered bindings.

Important Methods

- **discoverServices:** This method performs a discovery to find services bound to a specific binding or subbinding. The discovery is performed asynchronously. Therefore, the `result` is available not until the given `timeout` is elapsed. Further, a `ServiceAdvertisement` with omitted data is given as search parameter. The omitted data is the information to search for.

  ```
  public void discoverServices(DiscoveryResult result,
      ServiceAdvertisement searched, long timeout);
  ```

  - **result:** An interface that is informed about the `result` of the discovery (i.e., a set of discovered services) after the timeout has elapsed.
  - **searched:** The data used for the discovery.
  - **timeout:** A timeout in milliseconds to wait for the `result`.

2.6.2 ServiceAdvertisement

```
public class ServiceAdvertisement implements Serializable
```

A ServiceAdvertisement describes the contact data of a `Service`, including its service id, bindings, name, and the id of the node it is running on.

Constructors The only constructor is the default constructor, which creates an empty `ServiceAdvertisement`.

```
public ServiceAdvertisement(){...}
```

Important Members

- **id:** The id of the service.

  ```
  private String id;
  ```

- **nodeId:** The id of the node the service is running on.

  ```
  private String nodeId;
  ```

- **bindings:** A set of bindings the service has registered itself to.

  ```
  private Set<String> bindings;
  ```

- **name:** The human readable name of the service.

  ```
  private String name;
  ```

- **expirationDate:** Defines a date when the data in this `ServiceAdvertisement` might not hold anymore.[2] This time is based on the constant `TIME_TO_LIVE`.

  ```
  private Date expirationDate;
  ```

2.6.3 DiscoveryResult

```
public interface DiscoveryResult
```

After the timeout of the `DiscoveryResult` has elapsed, the `DiscoveryResult` is notified with the results of the discovery request. The interface has to be overridden by the application that issued the discovery request.

Important Methods

- **servicesDiscovered:** This method is called when the timeout of the discovery request has elapsed. It contains the result of that request.

  ```
  public void servicesDiscovered(
      List<ServiceAdvertisement> serviceAdvertisments);
  ```

- **isValid:** Returns whether the `TIME_TO_LIVE` has passed since the creation of this `ServiceAdvertisement`.

  ```
  public boolean isValid();
  ```

2.7 Trust Metric Infrastructure

Trust is the basis for cooperation within an organization or society. It influences an individual's thinking, behavior, and interactions. Thus, as in real life, trust is a key enabler for cooperation in software systems consisting of multiple interdependent entities, such as multi-agent systems (MAS) [10].

In MAS, autonomous proactive entities interact with each other in diverse ways to achieve a mutually favorable outcome. Just like in real life, interactions can be exploited for the benefit of only one party. One way to counter such detrimental behavior is to incorporate trust in the system and record the way an interaction partner behaved. If agents continuously interact with the same interaction partners, they can use their own interaction history (*direct trust*) to assess another agent. If interaction partners change often, a form of *reputation* [8] is beneficial which allows all agents to profit from the experiences of a few.

Trust is a complex concept that consists of multiple facets [14] such as credibility and reliability. Because agents can behave differently towards different agents, trust is subjective. Moreover, since agents fulfill different functions in a system and their behavior depends on external factors, an agent's role and the context of the interaction need to be regarded [15].

[2]Whenever a service is relocated to another node, a proxy is set up that temporarily forwards messages to the relocated service for a predefined amount of time. After that time, the proxy is shut down.

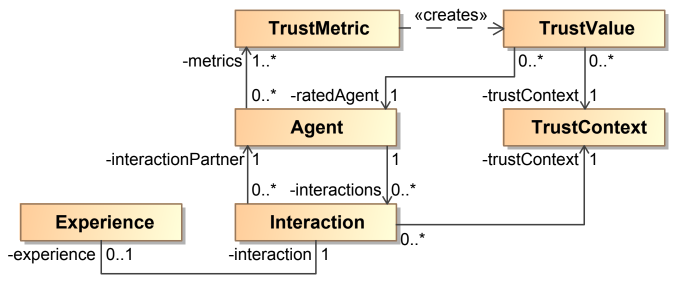

Figure 6: Concepts for the measurement and use of trust in a MAS [2]

Anders et al. [2] synthesize existing works on the software engineering aspects of trust and the usage of trust in MAS into *patterns* that cover various requirements in the very general setting of MAS. The patterns thus support a software engineer in implementing these well-established trust concepts into a MAS. From the body of literature and our their own experience, Anders et al. identified a set of key concepts necessary to measure and use trust in MAS, depicted in Figure 6. As mentioned before, trust in an `Agent` develops through repeated `Interactions` with it. If no interaction has taken place yet, initial trust [7] or reputation can be used. For each interaction, an agent can store its outcome as an `Experience`. Environmental factors as well as the roles of the interacting agents are captured in the `TrustContext`. This concept can comprise any factors that can change the behavior of an agent, such as the current time of day, the duration of the interaction, or the type of the interaction partner.

When one or more interactions have been completed and an agent wants to determine the trustworthiness of one of its interaction partners, a `TrustValue` for the interaction partner in a specific trust context is calculated using one of the `TrustMetrics`. Such metrics use the experiences with the interaction partner to determine a trust value and can be used to implement different trust models.

In some cases, the required calculations are complex and it might be necessary to *transform* the data collected in the experiences, e.g., to statistical data. These intermediate data can then be *interpreted* to yield a trust value. The TEM's Trust Metric Infrastructure, which is presented in this section, supports the transformation of measured data into intermediate data and the interpretation to a trust value. More precisely, it allows to store experiences, provides interfaces for metrics and trust contexts, and allows to derive trust values [6].

Depending on the concrete requirements of the modeled system, the metrics can also be specified in a way to be able to deal with some of the aspects of trust mentioned above. It might, e.g., be beneficial to discount older experiences and base the calculation of the trust value only on recent ones. Descriptions of different metrics that incorporate this aspect can be found in [5].

2.7.1 Architecture

The objective of the Trust-Enabling Middleware (TEM) is to collect trust values for improved self-x algorithms and provide trust information to the application level. The TEM provides techniques to handle trust operations for different trust contexts and is usable for measurements of different trust facets. It has to be noted that the TEM has also the ability to handle those trust values for facets even at runtime. Both trust values from direct observation and reputation are considered in this report. Trust from direct observation is derived from experiences gained in direct interactions with a communication partner. Reputation in contrast is a trust value derived from other parties' experiences. Reputation is especially of use if a potential interaction partner is not known.

The TEM provides an infrastructure to save trust related information and calculate trust data from this information in an easy and standardized way. The infrastructure presented here also allows to define application-specific trust metrics. The basic procedure that is used to derive trust data from a set of experiences, called raw data, is shown in Figure 7.

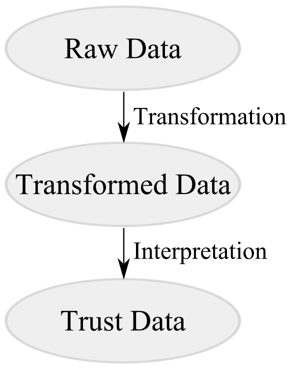

Figure 7: Multi-Stage Process: Retrieving Trust Data

Services gather `RawData` that represent a set of experiences that were gained in interactions with a specific interaction partner. `RawData` is the input for the calculation of `TrustData`, e.g., in the form of a trust value. Therefore, `RawData` is transformed by a `Transformer` into a preprocessed form called `TransformedData` that contains data relevant for the calculation of the `TrustData` (e.g., only the most current measurements might be of interest (sliding time window)). An `Interpreter` then derives the `TrustData` from the `TransformedData`. `TrustData` can either be a simple double value or a more complex object that, e.g., contains time series analysis data. Figure 8 shows a class diagram describing the *Trust Metric Infrastructure*, which is one of the main features of the TEM.

In general, experiences are application-specific, so the five main concepts (`RawData`, `Transformer`, `TransformedData`, `Interpreter`, `TrustData`) have to be implemented by the concrete application. However, the TEM provides default implementations for

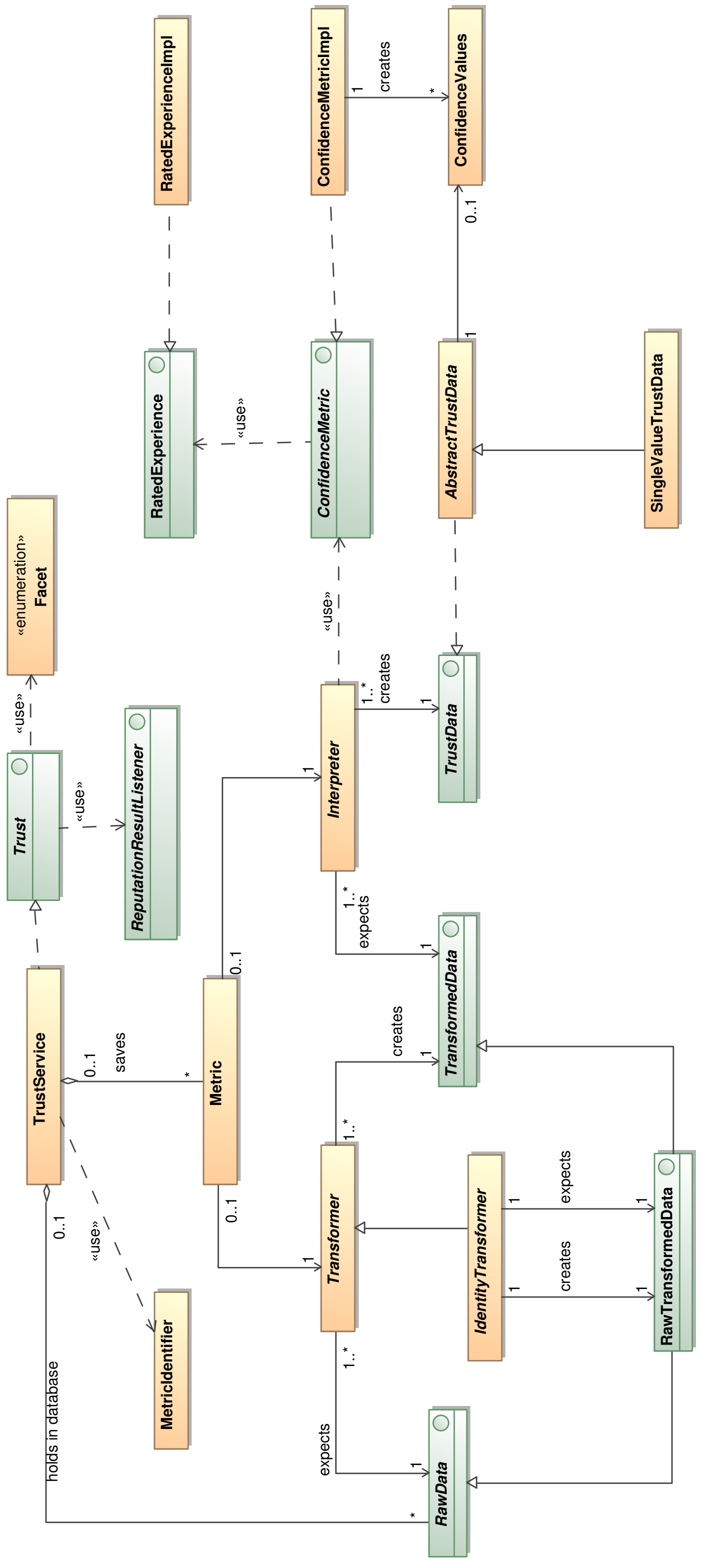

Figure 8: Trust Metric Infrastructure: Class Diagram

`Transformer` and `TrustData`. One of the transformers the TEM provides is the so-called `IdentityTransformer` that returns the given `RawData` as `TransformedData`. In this case, no `TransformedData` needs to be defined by the application. Instead, the corresponding `RawData` implementation has to use the interface `RawDataTransformed`. This class is a `RawData` as well as a `TransformedData`. The TEM also provides a `TrustData` implementation that only consists of a single double value, the `SingleValueTrustData`. Services can call methods to save and get `RawData`, specify the `Transformers` and `Interpreters` that are to be used when requesting `TrustData`, and start trust calculations. The TEM provides these methods to the `Services` in the form of an interface called `Trust`. A little example on how to use the Trust Metric Infrastructure can be found later in Section 2.7.6.

2.7.2 RawData

```
public interface RawData extends Serializable
```

`Raw Data` are experiences gathered by the services about their interaction partners. For example, the middleware gathers information about the reliability of other nodes in the form of `RawData` objects by observing message losses in order to estimate their reliability.

It is important to note that the own `RawData` implementation has to provide a default constructor. Otherwise, a `IncompatibleRawDataException` is thrown when calculating `TrustData` (see Section 2.7.6).

Important Methods

- **addRawData:** This method expects 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 context, facet, source as well as target node id and both of source and target service id or type, depending on situation. When calling the `addRawData` on the trust interface, the `addRawData` of the `RawData` class is called to merge the new data into the already existing one.

  ```
  public void addRawData(RawData newRawData);
  ```

- **deleteObsoleteData:** On the other hand, the method `deleteObsoleteData` is responsible for removing outdated `RawData` elements. This is especially useful to reduce the total amount of saved experiences when they tend to accumulate. Please note that this method can be implemented empty, if the functionality is not needed.

  ```
  public void deleteObsoleteData();
  ```

2.7.3 Transformer

```
public abstract class Transformer<R extends RawData, T extends TransformedData>
```

The Transformer transforms RawData into TransformedData. Transformer is an abstract class with two methods to override: getExpectedInputClass and transform (see below).

Generic Type Parameters

- **R:** Concrete RawData type used as input for the Transformer.

  ```
  R extends RawData
  ```

- **T:** Concrete TransformedData type used as output of the Transformer.

  ```
  T extends TransformedData
  ```

Important Methods

- **getExpectedInputClass:** This method returns the class of the RawData that is expected as input. Since Java removes generic information on runtime, this method needs to be implemented within all subclasses.

  ```
  public abstract Class<R> getExpectedInputClass();
  ```

- **transform:** Expects a RawData object and transforms it into TransformedData. This TransformedData contains the RawData in prepared form. TransformedData also needs to be implemented by the application. If RawData 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.

  ```
  protected abstract T transform(R rawData, String targetNodeId);
  ```

2.7.4 Interpreter

```
public abstract class Interpreter<T extends TransformedData, U extends TrustData>
```

The Interpreter interprets the TransformedData to yield TrustData. The abstract class Interpreter is similar to Transformer.

Generic Type Parameters

- **T:** Concrete TransformedData type used as input for the Interpreter.

  ```
  T extends TransformedData
  ```

- **U:** Concrete TrustData type used as output of the Interpreter.

  ```
  U extends TrustData
  ```

Important Methods

- **getExpectedInputClass:** Returns the class of the `TransformedData` that is expected as input. Since Java removes generic information on runtime, this method needs to be implemented of all subclasses.

  ```
  public abstract Class<T> getExpectedInputClass();
  ```

- **interpret:** Contains the logic to create from the given `TransformedData` an object of the class `TrustData`, which, e.g., represents a single double value.

  ```
  protected abstract U interpret(T transformedData,
      ConfidenceMetric confidenceMetric);
  ```

  The confidence metric will be explained later in Section 2.8 (`Confidence` interface). If no confidence is needed, the argument can be ignored.

2.7.5 TrustData

```
public interface TrustData extends Serializable
```

The `TrustData` interface should be implemented by any class that represents trust data. The node id will be automatically set by the `TrustService`.

Important Methods

- **getTargetNodeId:** This method returns the id of the target node.

  ```
  public String getTargetNodeId();
  ```

- **getTrustContext:** This method returns the `TrustContext` of the `TrustData`.

  ```
  public String getTrustContext();
  ```

- **getFacet:** This method returns the facet of the `TrustData`.

  ```
  public Facet getFacet();
  ```

- **getTargetServiceTypeOrId:** This method returns the id or type of the service whose `RawData` were used for the calculation or `null`, if node data were used.

  ```
  public String getTargetServiceTypeOrId();
  ```

- **getConfidence:** This method returns all confidence values of the trust value. If no values were set, a object with minus one values is returned.

  ```
  public ConfidenceValues getConfidence();
  ```

- **getTrustValue:** Returns the trust value as a single double value.

```
public double getTrustValue();
```

- **getSourceServiceTypeOrId:** This method returns the id or type of the service whose `RawData` were used for the trust calculation.

```
public String getSourceServiceTypeOrId();
```

2.7.6 Trust Interface

```
public interface Trust
```

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 `RawData`, and to trigger trust calculations. In the following, the methods are discussed in more detail.

Important Methods

- **addRawData, getRawData:** With `addRawData` the `RawData` gathered by the applications can be saved into the TEM. `RawData` are saved per context, facet, target node id and both source and target service id (the source node id is always the own node for direct trust). 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 `RawData`. The method `getRawData` accordingly returns the currently saved `RawData`.

```
public void addRawData(String sourceNodeId, String sourceServiceId,
    String targetNodeId, String targetServiceId, Facet facet,
    String trustContext, RawData rawData)
    throws IncompatibleRawDataException;

public List<RawData> getRawData(String sourceNodeId,
    String sourceServiceTypeOrId, String targetNodeId,
    String targetServiceTypeOrId, Facet facet,
    String trustContext);
```

  The attributes of the methods are discussed in the following:

  - **sourceNodeId:** The id of the node that made the experience. This can be different from this node in case of reputation. For direct trust, the source node is always this node.
  - **sourceServiceId:** The distinctive id of the service that made the interaction that resulted in this experience.
  - **targetNodeId:** The id of the node the experience was made with.

  - **targetServiceId:** The distinctive id of the service this experience was made with. This can be `null` if `RawData` about a node is saved.
  - **facet:** The facet.
  - **trustContext:** The trust context.
  - **rawData:** The `RawData` to save.
  - **sourceServiceTypeOrId:** The id of the service that saved the data or the type of service (all `RawData` objects from the similar service id or type will be returned) or `null` (all `RawData` independent who saved it relative to the other ids will be returned).
  - **targetServiceTypeOrId:** The id of the service the data is about, or its type (all `RawData` objects from the similar service id or type will be taken into account) or `null`, if data about a node should be returned.

- **calculateDirectTrust:** This method comes in two variants which can be used to calculate either trust in a node or in a service (see below). Both variants expect the context, facet, both of source and target service id or type, depending on the situation, and the node id of the node the `TrustData` should be calculated for.

```
public TrustData calculateDirectTrust(String sourceServiceTypeOrId,
    String targetServiceTypeOrId, String targetNodeId, Facet facet,
    String trustContext) throws IncompatibleRawDataException;

public <R extends RawData, T extends TransformedData, U extends TrustData>
    U calculateDirectTrust(String sourceServiceTypeOrId,
    String targetServiceTypeOrId, String targetNodeId, Facet facet,
    String trustContext, Transformer<R, T> transformer,
    Interpreter<T, U> interpreter) throws IncompatibleRawDataException;
```

  The `IncompatibleRawDataException` is thrown, if no `RawData` object can be dynamically created for the `Transformer`, e.g., it has no default constructor. In the simpler variant, no other arguments are needed and the `Transformer` and `Interpreter` are used, which are set by the `setDirectTrustMetric` method (see method below). If no direct trust metric is set, an `IllegalStateException` is thrown. The other version additionally expects the specific `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 simpler version that uses the default metric should suffice. If no `RawData` exists (i.e., `RawData` representing experiences that were made with the interaction partners), the metric cycle will be called with an empty list of `RawData`. Depending on the implementation of the trust metric, the returned `TrustData` then can, e.g., be an initial trust value.

- **setTrustMetric:** This method sets a default `Transformer` and `Interpreter` for a facet, service type 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.

```
public <R extends RawData, T extends TransformedData, U extends TrustData>
    void setTrustMetric(String serviceType, Facet facet,
    String trustContext, Transformer<R, T> transformer,
    Interpreter<T, U> interpreter);
```

- **calculateReputation:** This method calculates the reputation of one target interaction partner by directly asking all its neighbors (see [4]). For this purpose, direct trust values from all neighbors of the target interaction partner are gathered and locally combined to an overall value. The neighbors use the `Transformer` and `Interpreter` to calculate a direct trust value of target interaction partner. 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 partners 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.

```
public void calculateReputation(String sourceServiceTypeOrId,
    String targetServiceTypeOrId, String targetNodeId, Facet facet,
    String trustContext, long timeout,
    ReputationResultListener resultListener);
```

Usage The following example in form of a code fragment illustrates some of the concepts of this section and demonstrates an application that performs four main actions:

1. Create TEM nodes.
2. Create a new service object.
3. Show how a newly created service object can be registered and started on a given TEM node.
4. Perform direct trust calculation.

First we need an instance of the object `IdFactory` in order to generate ids for our `OcmNodes`.

```
IdFactory idFactory = new AscendingIdFactory();
```

This factory is then given as argument to the method `createLocalNode` that creates a new instances of the class `OcmNode`, namely `node1` and `node2`.

```
OcmNode node1 = OcmLocalFactory.createLocalNode(idFactory);
OcmNode node2 = OcmLocalFactory.createLocalNode(idFactory);
```

In order to extend our new generated nodes to TEM nodes with trust capabilities, we have to call the factory method `buildTEMNode` in the `TEMFactory`:

```
node1 = TEMFactory.buildTEMNode(node1);
node2 = TEMFactory.buildTEMNode(node2);
```

Then we create two dummy `Services`, called `dummy1` and `dummy2`.

```
DummyService dummy1 = new DummyService();
DummyService dummy2 = new DummyService();
```

By calling the method `registerService` and `startService`, the `Services` will be registered and started on the TEM node.

```
node1.registerService(dummy1);
node1.startService(dummy1);

node2.registerService(dummy2);
node2.startService(dummy2);
```

Assume now that `dummy1` is interacting with `dummy2` and wants to save its gained experiences with `dummy2`, i.e., `RawData`, into the Trust Metric Infrastructure. Therefore, the method `addRawData` of the `Trust` interface must be called which is obtained by the `TrustServiceConnector`.

```
// Creates new RawData objects with the had experiences
JUnitTestRawData rd1 = new JUnitTestRawData(1.0f);
JUnitTestRawData rd2 = new JUnitTestRawData(0.0f);
JUnitTestRawData rd3 = new JUnitTestRawData(0.3f);
JUnitTestRawData rd4 = new JUnitTestRawData(0.7f);

// Returns the trust interface for trust interactions
Trust trust1 = dummy1.getServiceConnector().getTrust();

// Merges the data in the given RawData with the RawData of this class
trust1.addRawData(node1.getId(), dummy1.getServiceId(), node2.getId(),
dummy2.getServiceId(), Facet.CREDIBILITY, null, rd1);

trust1.addRawData(node1.getId(), dummy1.getServiceId(), node2.getId(),
dummy2.getServiceId(), Facet.CREDIBILITY, null, rd2);

trust1.addRawData(node1.getId(), dummy1.getServiceId(), node2.getId(),
dummy2.getServiceId(), Facet.CREDIBILITY, null, rd3);

trust1.addRawData(node1.getId(), dummy1.getServiceId(), node2.getId(),
dummy2.getServiceId(), Facet.CREDIBILITY, null, rd4);
```

The function `getServiceTypeFromId` returns the type of a service from the given service id which is given as argument to the `setTrustMetric` function. If the trust metric is set, the method to calculate direct trust (i.e., `calculateDirectTrust`) can be called.

```
// Returns the type of a service from the given serviceId
String serviceType = Util.getServiceTypeFromId(dummy2.getServiceId());

// Sets a transformer and interpreter to use for the given service type,
// context and facet combination
trust1.setTrustMetric(serviceType, Facet.CREDIBILITY, null,
    new JUnitTransformerIdentity(), new JUnitInterpreterMean());

// Calculate direct trust
TrustData td = trust.calculateDirectTrust(dummy1.getServiceId(),
dummy2.getServiceId(), node2.getId(), Facet.CREDIBILITY, null);

// Access the trust value
double trustValue = td.getTrustValue();
```

2.8 Confidence

Agents can use trust values that characterize the behavior of their interaction partners as a measure of uncertainty, allowing the agents to make more appropriate decisions. However, because of the systems' dynamics and the agents' limited knowledge, the accuracy of these trust values is often limited as well, introducing another form of uncertainty. Thus, agents can make use of the *confidence in a trust value* [3], which indicates the degree of certainty that a trust value describes the actual observable behavior of an agent. While both direct trust or reputation can be used for agents to make decisions, confidence provides an estimation when to use direct trust or when to use reputation or can even be used to make a weighted decision. The confidence in a trust value depends on three different criteria:

- **Number of experiences:** The more experiences with an interaction partner were made, the more confidence we have in the trust value. This is the case since – if the interaction partner does not change its behavior in a severe way – more experiences mean that the trust value tends to describe the expected observable behavior more precisely. Outliers therefore do not carry that much weight, similar to the law of large numbers.

- **Age of experiences:** Because interaction partners may change their behavior between two interactions, the older the experiences with an interaction partner, the less confidence in the trust value. Thus, recent experiences should be considered more than older experiences as it is likely that they mirror more accurately the interaction partner's current behavior.

- **Variance of experiences:** The more variance in the interaction partner's behavior and thus in the experiences with it, the less confidence in the trust value, because the actual observable behavior is rather likely to differ from the expected behavior described by the trust value. This is especially the case when an interaction partner often changes its behavior.

The TEM provides the possibility to calculate the confidence in a trust value by using the already existing Trust Metric Infrastructure. The most important concepts for using the confidence are explained in the following.

2.8.1 RatedExperience

```
public interface RatedExperience
```

The confidence is calculated from a set of experiences. In the TEM, these experiences have to be instances of the `RatedExperience` interface. A `RatedExperience` provides information about an experience that is essential for the confidence metric to work.

Important Methods

- **getTimestamp:** This method returns the time (as a timestamp) when the experience occurred.

  ```
  public long getTimestamp();
  ```

- **getRating:** This method returns the rating of the experience, a value between 0 and 1.

  ```
  public double getRating();
  ```

2.8.2 ConfidenceValues

```
public class ConfidenceValues implements Serializable
```

The `ConfidenceValues` class contains the total confidence values as well as the values for the three confidence parts: number, age and variance. This class is returned when calling one of the `calculateConfidence` methods in the `ConfidenceMetric` interface (see Section 2.8.3). This class is immutable as all fields are final and set in the constructor. Therefore, only getter methods exist.

Important members

- **MINUS_ONE**: A static instance of the class, which contains only -1 values for all confidence values. This can be used, e.g., to be returned as confidence value from `TrustData` as an alternative to `null`.

  ```
  public static final ConfidenceValues MINUS_ONE
    = new ConfidenceValues(-1, -1, -1, -1);
  ```

- **confidence**: Contains the total confidence.

  ```
  private final double confidence;
  ```

- **numberConfidence**: Contains the number confidence.

  ```
  private final double numberConfidence;
  ```

- **ageConfidence**: Contains the age confidence.

  ```
  private final double ageConfidence;
  ```

- **varianceConfidence**: Contains the age confidence.

  ```
  private final double varianceConfidence;
  ```

Important methods

- **getConfidence**: Returns the total confidence.

  ```
  public double getConfidence();
  ```

- **getNumberConfidence**: Returns the number confidence.

  ```
  public double getNumberConfidence();
  ```

- **getAgeConfidence**: Returns the age confidence.

  ```
  public double getAgeConfidence();
  ```

- **getVarianceConfidence**: Returns the variance confidence.

  ```
  public double getVarianceConfidence();
  ```

2.8.3 ConfidenceMetric

```
public interface ConfidenceMetric
```

The confidence metric to calculate the confidence in a trust value is used through the `ConfidenceMetric` interface. This interface can be accessed through the `Interpreter` class and provides multiple methods which calculate the confidence depending on the number, age and variance of experiences.

Important Methods

- **calculateNumberConfidence:** This method comes in two variants (see below). Both variants expect a list of `RatedExperience`s.

  ```
  public abstract double calculateNumberConfidence(
      List<RatedExperience> experiences);

  public abstract double calculateNumberConfidence(
      List<RatedExperience> experiences, int numberThreshold);
  ```

  In the simple variant, no other arguments except `RatedExperience`s are needed and the `numberThreshold` (i.e., indicating, that enough experiences were gathered to assume a confidence of 1) is extracted from the configuration file which is called `tem.properties`. The more complex version expects the `numberThreshold` as additional argument. This version of the method can be used if the variable `numberThreshold` needs to be set at runtime.

- **calculateAgeConfidence:** This method has two different signatures (see below). Both signatures expect a list of `RatedExperience`s.

```
public abstract double calculateAgeConfidence(
    List<RatedExperience> experiences);

public abstract double calculateAgeConfidence(List<RatedExperience>
    experiences, long upToDateThreshold, long outdatedThreshold);
```

In the first signature, no other arguments except `RatedExperience`s are needed and the time based thresholds (i.e., `upToDateThreshold` and `outdatedThreshold` in milliseconds) are extracted from the configuration file. The other signature additionally expects both thresholds as argument. This type of the signature can be used if the time based thresholds need to be set at runtime.

- **calculateVarianceConfidence:** This method calculates the confidence based on the variance of the experiences the trust calculation was based upon. The additional weights list provides `weights` in case the trust calculation was based on a weighted mean metric. In this case a weighted variance is used (see below).

```
public abstract double calculateVarianceConfidence(
    List<RatedExperience> experiences, double trustValue);

public abstract double calculateVarianceConfidence(List<RatedExperience>
    experiences, List<Double> weights, double trustValue);
```

- **calculateConfidence:** This method has four different signatures. All of them expect a list of `RatedExperience`s as well as a `trustValue`. The first one (see below) is the most simple variant. It needs only experiences and a trust value as argument and reads all thresholds and weights from the configuration file.

```
public abstract ConfidenceValues calculateConfidence(
    List<RatedExperience> experiences, double trustValue);

public abstract ConfidenceValues calculateConfidence(
    List<RatedExperience> experiences, List<Double> weights,
      double trustValue);

public abstract ConfidenceValues calculateConfidence(
    List<RatedExperience> experiences, double trustValue,
    float numberConfidenceWeight, float ageConfidenceWeight,
    float varianceConfidenceWeight);

public abstract ConfidenceValues calculateConfidence(
    List<RatedExperience> experiences, List<Double> weights,
    double trustValue, float numberConfidenceWeight,
    float ageConfidenceWeight, float varianceConfidenceWeight);
```

The second one further expects a list of weights which is used as weighted mean to calculate the trust value. The last two signatures calculate confidence using the given thresholds which are discussed in the following:

  - **numberConfidenceWeight:** The weight of the number confidence value when calculating the total confidence.
  - **ageConfidenceWeight:** The weight of the age confidence value when calculating the total confidence.

  - **varianceConfidenceWeight:** The weight of the variance confidence value when calculating the total confidence.

- **calculateTotalConfidence:** This method calculates a total confidence using the given weights (see above). The weights can be any number and do not have to sum to one.

```java
public abstract double calculateTotalConfidence(double numberConfidence,
    double ageConfidence, double varianceConfidence,
    float numberConfidenceWeight, float ageConfidenceWeight,
    float varianceConfidenceWeight);
```

Usage As was shortly described in Section 2.7.4, the `ConfidenceMetric` interface is provided and can be used in the `interpret` method of the `Interpreter`. The TEM provides a default implementation for the `ConfidenceMetric` interface, which is the implementation of the interface given in the `interpret` method of the `Interpreter`. The metrics can be parametrized using the property file of the TEM, see Section 4.6. After calculating the `TrustData`, the confidence can be calculated by calling one of the `calculateConfidence` methods and can be saved in the `TrustData` class. If a specific implementation of the `ConfidenceMetric` is required, such an implementation can be instantiated and used in the `interpret` method instead of the provided one. In this case the applications has to be aware that a non default confidence metric implementation was used.

The method `getConfidence` of the `TrustData` has to return the calculated confidence values using the `ConfidenceValues` object. The method can return `null` in case no confidence values are calculated, but the applications using that `TrustData` object should be aware of that. The abstract class `AbstractTrustData` provides the field and respective setter method to save the calculated confidence values. The `getConfidence` method of `AbstractTrustData` returns the set confidence values or `null`, if the setter was not called.

3 Multi-Agent System Concepts

The TEMAS results from a combination of the TEM and some MAS-specific concepts that extend the TEM to a full-fledged MAS. These additional MAS-specific concepts are consolidated in the *MASConcepts4TEM* (see Figure 9). The MASConcepts4TEM primarily introduce the concept of agents (see Section 3.1), additional trust concepts such as *Trust-Based Scenarios* (see Section 3.4 and Section 3.5), and basic time concepts (see Section 3.6). With respect to the TEM, it further serves as a facade because it hides the complexity of the underlying infrastructure consisting of nodes and services and dependent interfaces to higher level applications. This results, e.g., in simpler, more common, and natural interfaces for messaging and the application of trust in MAS.

The TEMAS provides some functionality implemented in the form of `Agents` that can be used out of the box on the one hand, and some concepts that have to be extended and concretized depending on the individual application on the other hand. Concepts and functionality that can be used out of the box comprise the `InternalCalendar` (see Section 3.6.4) and the `YellowPages` (see Section 3.7). Application-specific agents can be implemented by either extending the classes `Agent` (see Section 3.1.4) or `TrustAgent` (see Section 3.5.2), depending whether or not the agent should be able to make trust-aware decisions.

To be able to debug and test TEMAS-based applications without introducing the complexity of the TEM's underlying infrastructure of nodes, system services, and processing overhead, the MASConcepts4TEM define a simple simulation environment called *temLight simulation environment* which is explained in Section 4.

The additional concepts defined by the MASConcepts4TEM are presented in the following subsections.

3.1 Agent Concepts

In this section, we present interfaces and classes that define an agent's basic properties, functionality, and capabilities.

3.1.1 IAgent

```
public interface IAgent extends ITimeProvider
```

An `IAgent` is an interface that defines an agent in its simplest form. Thus, it should be implemented by any class that represents an agent. The interface `IAgent` defines that each agent can be identified by a unique id, called agent identifier, and has a method, called `achieveGoals()` that is periodically called by the underlying MAS, allowing the `IAgent` to achieve its goals. Because each agent should be aware of the time, it extends the interface `ITimeProvider` (see Section 3.6.1).

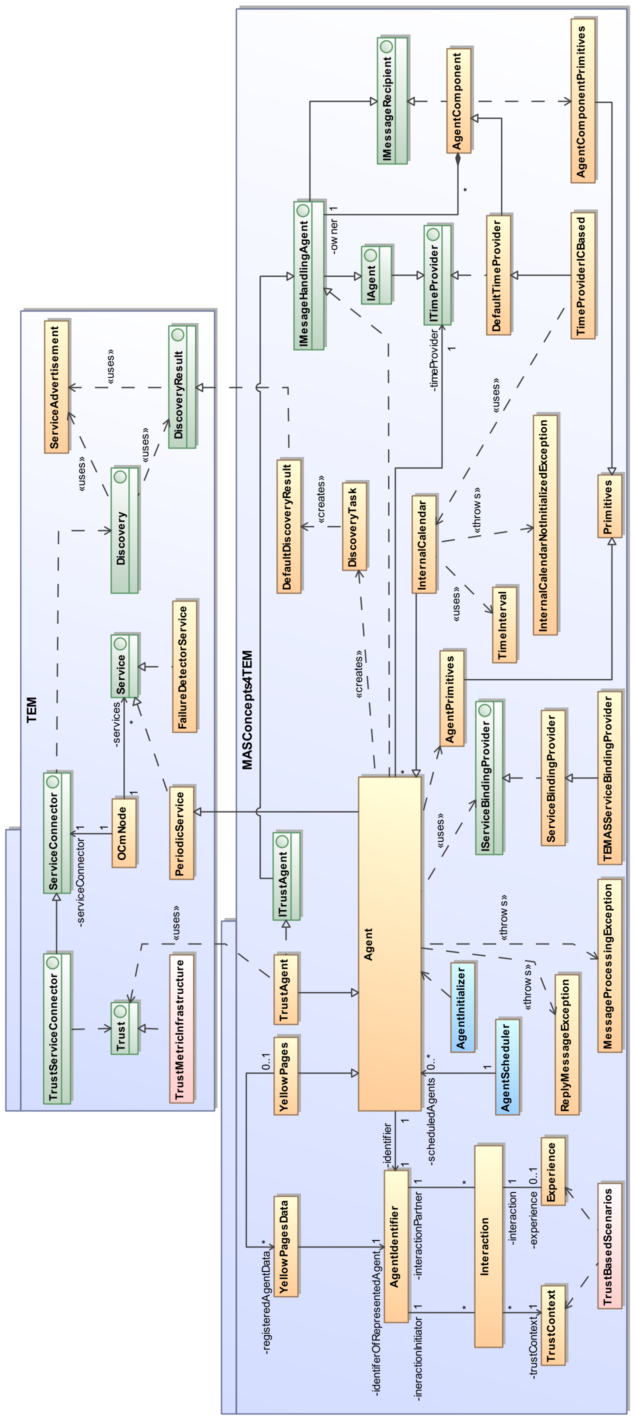

Figure 9: By defining additional concepts and functionality, the MASConcepts4TEM extend the TEM to the TEMAS. Here, the concept `TrustBasedScenarios` (highlighted in red) represents multiple concepts that implement the Trust-Based Scenarios' actual functionality (see Section 3.4.4). The concepts `AgentInitializer` and `AgentScheduler` (highlighted in blue) are introduced in Section 4.1 and Section 4.4.

Important Methods

- **getAgentIdentifier:** This method returns the agent's unique identifier, such as provided by the `AgentIdentifier` (see Section 3.1.5).

  ```
  public String getAgentIdentifier();
  ```

- **achieveGoals:** This method is periodically called by the underlying MAS and informs the agent to pursue its goals, i.e., this method triggers periodic actions performed by the agent. The frequency depends on the concrete implementation.

  ```
  public void achieveGoals();
  ```

Usage This interface is extended by the interface `IMessageHandlingAgent` (see Section 3.1.3), an `IAgent` which is able to send and receive messages.

3.1.2 IMessageRecipient

```
public interface IMessageRecipient
```

The `IMessageRecipient` is an interface that marks classes that are able to receive messages sent by `IMessageHandlingAgent`s. These classes do not necessarily have to be an agent, i.e., implement `IAgent`. It is mandatory that classes implementing this interface provide a method `public static Primitives getPrimitives()` that returns `Primitives` (see Section 3.2.1), defining what messages the `IMessageRecipient` can receive and process.

Usage This interface is extended by the interface `IMessageHandlingAgent` (see Section 3.1.3) and implemented by the class `AgentComponent` (see Section 3.1.6).

3.1.3 IMessageHandlingAgent

```
public interface IMessageHandlingAgent extends Serializable,
    IAgent, IMessageRecipient
```

An `IMessageHandlingAgent` defines an interface that is used for communication with other agents within the system. As each agent is able to send and receive messages in a common multi-agent system, this interface should be implemented by any class that represents an agent (that is why this interface extends the interfaces `IAgent` (see Section 3.1.1) and `IMessageRecipient` (see Section 3.1.2).

Important Methods

- **sendEvent:** Sends a message with primitive `thePrimitive` and payload `data` to an agent with identifier `agentIdentifier`. The sender does *not* expect the receiver to reply to this message. The method returns the message's unique identifier if it could be sent, or `null` if it could not be sent.

```
public UUID sendEvent(String agentIdentifier, String thePrimitive,
    Serializable data);
```

- **sendRequest:** Sends a message with primitive `thePrimitive` and payload `data` to an agent with identifier `agentIdentifier`. The sender demands that the receiver responds to this message. Again, the method returns the message's unique identifier if it could be sent, or `null` if it could not be sent.

```
public UUID sendRequest(String agentIdentifier, String thePrimitive,
    Serializable data);
```

- **waitForReply:** This method waits `timeout` ms for a reply to a previously sent message with identifier `messageIdentifier` and primitive `thePrimitive` from an `IMessageHandlingAgent` with identifier `agentIdentifier`. If the reply is not received within the specified time, a `java.util.concurrent.TimeoutException` (see Java documentation) is thrown. The specified `messageIdentifier` unambiguously identifies the reply message. If an exception was thrown while processing the message that should trigger the reply, this method throws a `RuntimeException` that encapsulates the exception's cause.

```
public Serializable waitForReply(UUID messageIdentifier,
    String agentIdentifier, String thePrimitive, int timeout)
    throws TimeoutException, RuntimeException;
```

- **sendRequestAndWaitForReply:** This method is a combination of the calls `sendRequest(...)` and `waitForReply(...)`. It sends a message with primitive `thePrimitive` and payload `data` to an agent with identifier `agentIdentifier`. Afterwards, the method waits `timeout` ms for a reply. Again, if the reply is not received within the specified time, a `TimeoutException` is thrown, and if an exception was thrown while processing the message, this method throws a `RuntimeException` that encapsulates the exception's cause.

```
public Serializable sendRequestAndWaitForReply(String agentIdentifier,
    String thePrimitive, Serializable data, int timeout)
    throws TimeoutException, RuntimeException;
```

- **broadcastEvent:** This method is similar to `sendEvent(...)` but addresses multiple receivers. It broadcasts a message with primitive `thePrimitive` and payload `data` to all agents that are bound to the service binding `typeOfReceiver`. The sender does *not* expect the receivers to reply to this message. As with `sendEvent(...)`, the method returns the message's unique identifier if it could be sent, or `null` if it could not be sent.

```
public UUID broadcastEvent(String typeOfReceiver, String thePrimitive,
    Serializable data);
```

- **broadcastRequest:** Sends a message with primitive `thePrimitive` and payload `data` to all agents that are bound to the service binding `typeOfReceiver`. The sender demands that the receivers respond to this message. However, since it is not known how many agents actually receive this message, only one receiver has to send a response. Note that, even if more than one receiver replies to the broadcast, the sender can only process one reply by calling `waitForReply(...)` afterwards. All other replies are discarded. Again, the method returns the message's unique identifier if it could be sent, or `null` if it could not be sent.

  ```
  public UUID broadcastRequest(String typeOfReceiver, String thePrimitive,
      Serializable data);
  ```

- **broadcastRequestAndWaitForReply:** This method is a combination of the calls `broadcastEvent(...)` and `waitForReply(...)`. It broadcasts a message with primitive `thePrimitive` and payload `data` to all agents that are bound to the service binding `typeOfReceiver`. Afterwards, the method waits `timeout` milliseconds for a reply. Again, if the reply is not received within the specified time, a `TimeoutException` is thrown, and if an exception was thrown while processing the message, this method throws a `RuntimeException` that encapsulates the exception's cause. Only a single reply is returned[3]. All other replies are discarded. If multiple replies are received, the reply that is returned is randomly chosen.

  ```
  public Serializable broadcastRequestAndWaitForReply(String typeOfReceiver,
      String thePrimitive, Serializable data, int timeout)
      throws TimeoutException, RuntimeException;
  ```

Usage This interface is implemented by the class `Agent` (see Section 3.1.4) and extended by `ITrustAgent` (see Section 3.5.1).

3.1.4 Agent

```
public abstract class Agent extends PeriodicService
    implements IMessageHandlingAgent
```

This abstract class represents an agent in the MAS that is run on a `OCmNode` (see Section 2.1). It is thus a concrete implementation of the TEM's `PeriodicService` (see Section 2.4) and further provides additional methods to request the current time and to simplify message handling by implementing the `IMessageHandlingAgent` interface. Although the TEMAS hides the complexity of the underlying infrastructure, the `Agent` can provide information about the `OCmNode` it runs on. Note that basic `Agents` cannot make use of the TEM's Trust Metric Infrastructure (see Section 2.7). If an agent should be able to make use of the Trust Metric Infrastructure, it has to extend the `TrustAgent` (see Section 3.5.2).

[3]While it is undefined which reply is returned, usually the first reply that is received should be returned

Constructors Creates a new `Agent` and an instance of the given `timeProviderClass` used to retrieve the current time. The `serviceBindingProvider` (see Section 3.2.9) can be used later on to generate and query the service bindings of specific types of `Agents`. In general, service bindings specify the agent or a group of agents that should receive a message.

```
public Agent(ServiceBindingProvider serviceBindingProvider,
    Class<? extends DefaultTimeProvider> timeProviderClass) {...}
```

Important Members

- **identifier:** A unique identifier that unambiguously identifies this agent (see Section 3.1.5).

  ```
  protected AgentIdentifier identifier;
  ```

- **timeProvider:** The `DefaultTimeProvider` from which the current time is retrieved (see Section 3.6.2).

  ```
  private DefaultTimeProvider timeProvider;
  ```

- **SIMULATE_TEM_MESSAGE_TRANSPORT:** Switches serialization and deserialization of data sent via messages on or off. This should only be activated if the application is deployed in the temLight simulation environment (see Section 4).

  ```
  public static final boolean SIMULATE_TEM_MESSAGE_TRANSPORT = false;
  ```

- **DEFAULT_TIMEOUT:** A predefined amount of time (in ms) `Agents` wait for reply messages.

  ```
  public static final int DEFAULT_TIMEOUT = 5000;
  ```

Important Methods

- **initAgent:** Initializes an `Agent` with the given data, which might be `null` if no data is necessary. If a customized initialization of an `Agent` implementation is necessary, this method must be overridden.

  ```
  public void initAgent(Map<String, Serializable> initData) {...}
  ```

- **achieveGoals:** The method periodically called by the underlying MAS that informs the agent to pursue its goals. For example, in the TEM execution environment (see Section 4), this method is triggered when the `PeriodicService`'s `step()` method is called (see Section 2.4). Consequently, all actions that should be periodically performed by this `Agent` should be called in this method in the right order. This method is declared in the interface `IAgent` (see Section 3.1.1).

```
public void achieveGoals() {...}
```

Because it is not ensured that the default implementation of `achieveGoals()` is empty, subclasses of `Agent` should always call `super.achieveGoals()` first:

```
public class MyAgent extends Agent {
  ...

  /**
   * Overriding the achieveGoals method in a subclass of Agent.
   */
  public void achieveGoals() {
      // call super method first
      super.achieveGoals();

      // call MyAgent-specific methods afterwards
      ...
  }
}
```

- **getPrimitives:** A static method that gets the `Agent`'s primitives, which define the messages the `Agent` can receive. By default, this is an instance of `AgentPrimitives`. This method is similar to the `getPrimitives()` method declared in the class `AgentComponent` (see Section 3.1.6).

  ```
  public static AgentPrimitives getPrimitives() {...}
  ```

  Subclasses of `Agent` can specify a subclass of `AgentPrimitives` and refine the `getPrimitives()` method. If the primitives of an imaginary class `MyAgent` are defined in an imaginary class `MyAgentPrimitives`, `MyAgent` refines `getPrimitives()` as follows:

  ```
  public class MyAgent extends Agent {

    ...

    /**
     * A refined getPrimitives() method defined in an imaginary subclass of
     * Agent called MyAgent.
     * MyAgent can receive messages with primitives specified in the classes
     * MyAgentPrimitives and AgentPrimitives.
     */
    public static MyAgentPrimitives getPrimitives() {
        return new MyAgentPrimitives(MyAgent.class);
    }
  }
  ```

- **getAgentIdentifier:** Gets the unique `identifier` in the form of a string that unambiguously identifies this `Agent` (see Section 3.1.5).

  ```
  public String getAgentIdentifier() {...}
  ```

- **getActualAgentIdentifier:** Gets, in contrast to `getAgentIdentifier(...)`, the `Agent`'s unique `identifier` in the form of an `AgentIdentifier` (see Section 3.1.5).

```
public AgentIdentifier getActualAgentIdentifier() {...}
```

- **getCurrentTime:** Returns the current time in form of a `GregorianCalendar` (see Java documentation). The time is requested from the `Agent`'s `timeProvider` (see Section 3.6.2).

  ```
  public GregorianCalendar getCurrentTime() {...}
  ```

- **getInterval:** Returns the time in milliseconds for the interval between the calls of the `step()` method triggered by the TEM. This value is only regarded when using the asynchronous execution model in the TEM execution environment (see Section 4.1 and Section 4.3).

  ```
  protected long getInterval() {...}
  ```

- **getServiceId:** Gets the service id of the service represented by this agent (see Section 3.1.5).

  ```
  public String getServiceId() {...}
  ```

- **getServiceType:** Returns the service type of this service (`Agents` implement the interface `Service`, presented in Section 2.2). The service type is a part of the service id. By default, the service type of all agents is the Java fully qualified class name of the class `Agent` (see method `getServiceTypeForClass`).

  ```
  public String getServiceType() {...}
  ```

- **getServiceTypeForClass:** Returns the service type of this `Agent`. By default, this is the Java fully qualified class name of the class `Agent`.

  ```
  public static String getServiceTypeForClass() {...}
  ```

- **getNodeId:** Gets the identifier of the `OcmNode` that currently hosts this agent (see Section 3.1.5).

  ```
  protected String getNodeId() {...}
  ```

- **discoverAgents:** This method uses the `Discovery` functionality of OCµ (see Section 2.6). Calling this method starts a discovery of `Agents` that are bound to at least one of the given service bindings. Note that this method might not find all `Agents` in the system that are bound to one of the service bindings if the TEM simulation environment is used (see Section 4). Waits at most `timeout` milliseconds for the results. The method returns a `DiscoveryTask` (see Section 3.3.1) running in a separate thread that returns the result in the form of a `DefaultDiscoveryResult` (see Section 3.3.2) if finished. The `DefaultDiscoveryResult` can be evaluated with the method `evaluateDiscoveryResult` below.

```
protected DiscoveryTask discoverAgents(int timeout,
    String... serviceBindings) {...}
```

- **evaluateDiscoveryResult:** Evaluates a `DefaultDiscoveryResult` by extracting and returning a set of `AgentIdentifiers` of `Agents` that have been identified in the course of a discovery. For an example how to use this and the method above, see Section 3.3.

```
protected Set<AgentIdentifier> evaluateDiscoveryResult(
    DefaultDiscoveryResult discoveryResult) {...}
```

- **sendEvent, sendRequest, waitForReply, sendRequestAndWaitForReply, broadcastEvent, broadcastRequest, broadcastRequestAndWaitForReply:** These methods implement the messaging functionality (see Section 3.1.3) on the basis of OCµ's `MessageSender` and `EventMessages`. Note that the `Agent` throws, in contrast to the `IMessageHandlingAgent`, a `ReplyMessageException` instead of a common `RuntimeException`.

```
public UUID sendEvent(String agentIdentifier, String thePrimitive,
    Serializable data) {...}

public UUID sendRequest(String agentIdentifier, String thePrimitive,
    Serializable data) {...}

public Serializable waitForReply(UUID messageIdentifier,
    String agentIdentifier, String thePrimitive, int timeout)
    throws TimeoutException, ReplyMessageException {...}

public Serializable sendRequestAndWaitForReply(String agentIdentifier,
    String thePrimitive, Serializable data, int timeout)
    throws TimeoutException, ReplyMessageException {...}

public UUID broadcastEvent(String typeOfReceiver, String thePrimitive,
    Serializable data) {...}

public UUID broadcastRequest(String typeOfReceiver, String thePrimitive,
    Serializable data) {...}

public Serializable broadcastRequestAndWaitForReply(String typeOfReceiver,
    String thePrimitive, Serializable data, int timeout)
    throws TimeoutException, ReplyMessageException {...}
```

Usage The `Agent` is the major concept of the TEMAS. If you want to implement a trust-aware `Agent`, extend the concept `TrustAgent` (see Section 3.5.2).

3.1.5 AgentIdentifier

```
public class AgentIdentifier implements Cloneable
```

Each `Agent` can be uniquely identified by its `AgentIdentifier`. This identifier consists of the `Agent`'s type (that is, by default, its class name), its service id, and its node id. Note that two `AgentIdentifiers` are equal if and only if the service id of both `AgentIdentifiers` is equal (the type and node id is not regarded) so that an `Agent`'s identity does not change if it is moved from one `OcmNode` to another. The type is not needed to check for equality as the service id unambiguously identifies an `Agent`. The `Agent`'s `AgentIdentifier` is automatically created and set in its `init(...)` method. An `AgentIdentifier`'s `String` representation, which should be sufficient in most cases, can be requested by calling the `Agent`'s method `getAgentIdentifier()`. However, the class `AgentIdentifier` also provides a `static` method to create an `AgentIdentifier` object using a given `String` representation.

Constructors Creates a new instance of `AgentIdentifier` on the basis of the given `type`, `serviceId`, and `nodeId`.

```
public AgentIdentifier(String type, String serviceId, String nodeId) {...}
```

Important Members

- **type:** Holds the type (i.e., the Java fully qualified class name) of the `Agent` represented by this `AgentIdentifier`.

  ```
  private final String type;
  ```

- **serviceId:** Holds the service id of the `Agent` represented by this `AgentIdentifier`. The service id unambiguously identifies an `Agent`.

  ```
  private final String serviceId;
  ```

- **nodeId:** Holds the node id of the `Agent` represented by this `AgentIdentifier`. As an `Agent` might migrate from one `OcmNode` to another, the node id is not used to check for equality. It is thus not guaranteed that an `AgentIdentifier`'s node id is always up to date.

  ```
  private final String nodeId;
  ```

- **TYPE_SEPARATOR:** A special `String` sequence used to separate the `type` of the `AgentIdentifier` from its `serviceId` and `nodeId` in its `String` representation.

  ```
  private static final String AGENT_IDENTIFIER_SEPARATOR;
  ```

- **AGENT_IDENTIFIER_SEPARATOR:** A special `String` sequence used to separate the `AgentIdentifier`'s `serviceId` from its `nodeId` in its `String` representation.

  ```
  private static final String TYPE_SEPARATOR;
  ```

Important Methods

- **getType:** Returns the `AgentIdentifier`'s `type`.

  ```
  public String getType() {...}
  ```

- **getServiceId:** Returns the `AgentIdentifier`'s `serviceId`.

  ```
  public String getServiceId() {...}
  ```

- **getNodeId:** Returns the `AgentIdentifier`'s `nodeId`.

  ```
  public String getNodeId() {...}
  ```

- **createAgentIdentifier:** Creates a new `AgentIdentifier` from the given `String` representation of an `AgentIdentifier` (as returned by the `toString()` method in this class).

  If the `String` representation is not a valid representation of an `AgentIdentifier`, an `IllegalArgumentException` is thrown.

  ```
  public static AgentIdentifier createAgentIdentifier(String
      stringRepresentation) {...}
  ```

- **toString:** Returns a `String` representation of this `AgentIdentifier` in the following form:

  ```
  public String toString() {
      return this.type + TYPE_SEPARATOR + this.serviceId +
          AGENT_IDENTIFIER_SEPARATOR + this.nodeId;
  }
  ```

Usage An `Agent`'s `AgentIdentifier` can be requested by calling the `Agent`'s method `getActualAgentIdentifier()`. The `String` representation of this `AgentIdentifier` can be requested by calling the `Agent`'s `getAgentIdentifier()` method. If such a `String` should be used to create an `AgentIdentifier`, `createAgentIdentifier(...)` is to be called.

3.1.6 AgentComponent

```
public abstract class AgentComponent<T extends IMessageHandlingAgent>
    implements IMessageRecipient
```

The `AgentComponent` is an abstract class that represents and encapsulates a special capability that can be incorporated by `IMessageHandlingAgents` to extend their default capabilities such as messaging. An `IMessageHandlingAgent` can incorporate the capability defined by an `AgentComponent` by creating an instance of the `AgentComponent`. The agent can then use this capability by calling the `AgentComponent`'s methods that implement the functionality. Because `AgentComponents` might have to access information from the agent that incorporates the represented functionality, each AgentComponent is associated with the agent that holds it. This agent is called the AgentComponent's owner. To be able to access agent-specific information, the owner's type is parameterized. Importantly, AgentComponents can send and receive messages to and from other AgentComponents or IMessageHandlingAgents. The class AgentComponent therefore implements the interface IMessageRecipient. An AgentComponent can send messages via its owner, i.e., it calls its owner's methods to send messages and wait for replies. If an AgentComponent should be able to receive and react to messages, a concrete implementation of AgentComponentPrimitives is needed (see Section 3.2.3). This implementation defines the messages the corresponding AgentComponent can receive. If a concrete implementation of the AgentComponentPrimitives class exists, these primitives can be automatically registered with its owner when instantiating the corresponding AgentComponent. If the owner receives a message with a primitive one of its AgentComponents is registered for, it "redirects" this message to the corresponding AgentComponent. Further details are given in Section 3.2.1.

Generic Type Parameters

- **T:** The type of the AgentComponent's owner so that type-specific information can be retrieved.

```
T extends IMessageHandlingAgent
```

Constructors

- **public AgentComponent(T owner):** Creates an AgentComponent, associates it with its owner, and automatically registers the AgentComponent for its primitives with the owner if a corresponding implementation of AgentComponentPrimitives is available.

```
public AgentComponent(T owner) {...}
```

- **public AgentComponent(T owner, boolean registerWithOwner):** Here, the user can specify whether or not the AgentComponent and its primitives should be registered with the owner. If registerWithOwner is *true*, the AgentComponent is automatically registered with its owner.

```
public AgentComponent(T owner, boolean registerWithOwner) {...}
```

Important Members

- **T owner:** The IMessageHandlingAgent that owns the AgentComponent and incorporates its functionality. T specifies the concrete type of the owner so that the AgentComponent can access agent-specific methods.

```
private T owner;
```

Important Methods

- **public T getOwner():** Gets the `AgentComponent`'s owner.

  ```
  public T getOwner() {...}
  ```

- **public static AgentComponentPrimitives getPrimitives():** This is a static method that gets the `AgentComponent`'s primitives, i.e., a concrete implementation of `AgentComponentPrimitives`, which define the messages the `AgentComponent` can receive. By default, an `AgentComponent` can **not** receive specific messages, so that `null` is returned.

  ```
  /**
   * The default getPrimitives()-Method defined in AgentComponent.
   */
  public static AgentComponentPrimitives getPrimitives() {
      return null;
  }
  ```

  Subclasses of `AgentComponent`, however, can specify a subclass of the abstract class `AgentComponentPrimitives` and "refine"[4] the `getPrimitives()` method. For example, in case the primitives of a class `MyComponent` are defined in a class `MyPrimitives`, `MyComponent` "refines" `getPrimitives()` as follows:

  ```
  public class MyComponent extends AgentComponent<Agent> {

    ...

    /**
     * A refined getPrimitives() method defined in a subclass of
     * AgentComponent called MyComponent.
     * MyComponent can receive messages with primitives specified in the
     * class MyPrimitives.
     */
    public static MyPrimitives getPrimitives() {
        return new MyPrimitives(MyComponent.class);
    }
  }
  ```

  If an `AgentComponent` can not receive messages, the refined `getPrimitives()` method should return `null`. If this `AgentComponent` extends another concrete implementation of `AgentComponent`, this has to be done explicitly as otherwise the `AgentComponentPrimitives` of the superclass are returned.

Usage Should be extended by all special capabilities of an `Agent`. For instance, the `DefaultTimeProvider` is a subclass of `AgentComponent` (see Section 3.6.2).

[4]Since `getPrimitives()` is a static method, subclasses just hide the `getPrimitives()` method of their superclass.

3.2 Sending, Receiving, Processing, and Replying to Messages

Each `Agent` is able to send, receive, process, and reply to messages. This functionality is based on the messaging capabilities of the TEM (see Section 2.3). However, the `Agent` hides the complexity of messaging on the TEM layer to upper layers by implementing the `IMessageHandlingAgent` interface, which provides – from the perspective of MAS – a more common and natural interface. In the following, we introduce the `Primitives` concept, which is used to define which messages an `Agent` or `AgentComponent` is able to process on the one hand, and what the sender asks the message's receiver to do on the other hand. Further, we discuss technical details worth knowing to understand the `Agent`'s behavior when dealing with messages and to support debugging.

3.2.1 Primitives

```
public abstract class Primitives
```

An `IMessageRecipient`'s `Primitives` define the messages it can receive and process. While an `IMessageRecipient` can basically receive messages with arbitrary primitives, "can receive a message" means in this context that the `IMessageRecipient` knows the sender's intention, i.e., how to react to the message. Due to the fact that messages are processed by using a primitive string to identify the method that is to be called by the message's recipient by using Java reflection, each subclass of `Primitives` acts as a container holding and defining these primitive strings. To enable a specific `IMessageRecipient`, i.e., a specific subclass of `Agent` or `AgentComponent`, to process specific messages and to outline the messages it can react to to potential interaction partners as well as the programmer, usually each subclass of `Agent` and `AgentComponent` needs to define a corresponding subclass of `Primitives`. To establish a clear naming convention, `Primitives` classes should be named according to the corresponding `IMessageRecipient`. For example, primitives of the class `Agent` are collected in the `Primitives` class `AgentPrimitives`, and primitives of the class `AgentComponent` are defined in `AgentComponentPrimitives`. To simplify programming with respect to messages, the possible messages an `Agent` or `AgentComponent` can process are listed by calling the following method:

```
// T extends AgentPrimitives or AgentComponentPrimitives
public static T getPrimitives() {
    // A is the IMessageRecipient whose primitives are listed in T
    // and thus either of type Agent or AgentComponent
    return new T(A.class);
}
```

The static method `getPrimitives()` is to be implemented – and thus some kind of "overridden" – in each instance of `IMessageRecipient`[5]. It returns a new instance of the corresponding `Primitives` class, containing all primitive strings of messages that can

[5]Note that this is not mandatory if a subclass does not modify the set of messages its direct superclass can react to.

be processed by the IMessageRecipient. Obviously, as Agents are allowed to incorporate multiple AgentComponents, it should not be mandatory that the names of methods declared in incorporated AgentComponents or the Agent are unique (recall that we use Java reflection to react to incoming messages). Consequently, the IMessageRecipient that should process a specific message must be unambiguously identified. To avoid such name resolution conflicts, each Primitives class holds a prefix initialized in its constructor. This prefix specifies the name of the class of the corresponding IMessageRecipient (see example of getPrimitives() above). To clarify the constellation and outline how primitive strings look like, take a look at the following example:

```
// Message receiver:
public class MyReceiverAgent extends Agent {
    ...
    // This method is to be called via messages.
    // Note that this method must be public!
    public void receiverMethod() {...}
    ...
    // Gets the Primitives class of the MyReceiverAgent
    public static MyReceiverAgentPrimitives getPrimitives() {
        return new MyReceiverAgentPrimitives(MyReceiverAgent.class);
    }
}

// Message receivers primitive class:
public class MyReceiverAgentPrimitives extends AgentPrimitives {
    ...
    // prefix here is MyReceiverAgent.class.getName() + "."
    public final String RECEIVER_METHOD = this.prefix + "receiverMethod";
    ...
}

// Message sender:
public class MySenderAgent extends Agent {
    ...
    public void anyMethod() {
        ...
        // The AgentIdentifier of the message's recipient sent below
        AgentIdentifier myReceiverAgentId = ...;

        // MySenderAgent sends an event to MyReceiverAgent that triggers
        // a call of MyReceiverAgent's receiverMethod.
        // The call "MyReceiverAgent.getPrimitives()" returns a Primitives object
        // that includes all primitives MyReceiverAgent can react to.
        this.sendEvent(myReceiverAgentId,
            MyReceiverAgent.getPrimitives().RECEIVER_METHOD, null);
        ...
    }
    ...
}
```

The primitive string used by the message's sender to address and by the message's recipient to identify the correct method is encoded in the following String:

```
public final String RECEIVER_METHOD = this.prefix + "receiverMethod";
```

The String thus consists of MyReceiverAgent's fully qualified class name concatenated with "receiverMethod". Consequently, a primitive string starts with the type of the IMessageRecipient and ends with the method that should be called via Java reflection.

This method has to be publicly accessible. By establishing this convention, `Agents` are able to receive messages and identify the addressed methods correctly even if they hold different `AgentComponents` with equally named methods. To increase code flexibility and to avoid redundant code, the specific `Primitives` classes should reflect the class hierarchy of the corresponding `IMessageRecipients`. Following this convention avoids inconsistencies and simplifies code refactoring. The idea is that every specific subclass of `Primitives` automatically contains the primitive strings of its superclass and only has to declare new primitive strings for methods that should be additionally enabled for messaging. As a consequence, each `IMessageRecipient` is able to receive messages addressing methods that are declared by one of its superclasses by also inheriting the corresponding primitives. The following example demonstrates the `Primitives` hierarchy concept:

```java
public class AgentA extends Agent {
    ...
    public void methodOfA() {...}
    ...
    // Gets the Primitives class of AgentA
    public static AgentAPrimitives getPrimitives() {
        return new AgentAPrimitives(AgentA.class);
    }
}

// A subclass of AgentA
public class AgentB extends AgentA {
    ...
    public void methodOfB() {...}
    ...
    // Gets the Primitives class of AgentB
    public static AgentBPrimitives getPrimitives() {
        return new AgentBPrimitives(AgentB.class);
    }
}

public class AgentAPrimitives extends AgentPrimitives {
    ...
    // all Primitives of Agent are inherited
    public final String METHOD_OF_A = this.prefix + "methodOfA";
    ...
}

// A subclass of AgentAPrimitives:
// inherits primitives declared in AgentAPrimitives
public class AgentBPrimitives extends AgentAPrimitives {
    ...
    // METHOD_OF_A is inherited, only the new method has to be added
    public final String METHOD_OF_B = this.prefix + "methodOfB";
    ...
}
```

Because `Primitives` declare all messages that can be processed by an `Agent` or an `AgentComponent`, programming message-based interactions gets more intuitive and robust. When an `IMessageRecipient`-specific `Primitives` object is created by calling the corresponding `IMessageRecipient`'s method `getPrimitives()`, all possible messages that can be processed by the `IMessageRecipient` are shown to the programmer.

Constructors Creates a new instance of a concrete `Primitives` class that holds all declared primitive strings as static entries. The given class `theClass` initializes the prefix of each primitive string with the Java fully qualified class name of `theClass`.

```
public Primitives(Class<?> theClass) {...}
```

Important Members

- **prefix:** The unique prefix of the `Primitives` class, enabling unique primitive strings even if different `IMessageRecipients` feature equally named methods.

  ```
  protected final String prefix;
  ```

3.2.2 AgentPrimitives

```
public abstract class AgentPrimitives extends Primitives
```

`AgentPrimitives` is a subclass of `Primitives` (see Section 3.2.1). It declares the messages that can be processed by a generic `Agent`. For every specialization of `Agent`, a special `Primitives` class has to be created that extends the class `AgentPrimitives`. In these classes, additional primitive strings can be declared. As mentioned in Section 3.2.1, the agent-specific `AgentPrimitives` classes should reflect the class hierarchy of the corresponding `Agents`. Adhering to this convention avoids redundant code and enables `Agents` to receive and process messages addressing methods that are declared within their superclasses.

Constructors Creates a new instance of `AgentPrimitives` that defines all message primitives the corresponding `Agent` is able to process. The given class `theAgent` initializes the prefix with the Java fully qualified class name of `theAgent`.

```
public AgentPrimitives(Class<?> theAgent) {...}
```

Important Members Although the class `AgentPrimitives` defines two generic primitives, namely `ENABLE_STEP_METHOD` and `ACHIEVE_GOALS`, these primitives should not be used by the programmer.

3.2.3 AgentComponentPrimitives

```
public abstract class AgentComponentPrimitives extends Primitives
```

Like `AgentPrimitives`, `AgentComponentPrimitives` is a subclass of `Primitives` (see Section 3.2.1). The class `AgentComponentPrimitives` declares the messages that can be processed by a generic `AgentComponent`. For every specialization of `AgentComponent`, an `AgentComponent`-specific subclass of `AgentComponentPrimitives` has to be created. In these classes, additional primitive strings can be declared that extend the range of messages that can be processed by the AgentComponent. As mentioned in Section 3.2.1 and Section 3.2.2, AgentComponent-specific AgentComponentPrimitives classes should reflect the class hierarchy of corresponding AgentComponents. This avoids redundant code and enables AgentComponents to receive and process messages addressing methods that are declared within their superclasses.

Constructors Creates a new instance of AgentComponentPrimitives that defines all message primitives the corresponding AgentComponent is able to process. The given class theComponent initializes the prefix with the Java fully qualified class name of theComponent.

```
public AgentComponentPrimitives(Class<?> theComponent) {...}
```

3.2.4 Registering Primitives

As mentioned in Section 3.2.1, the primitives defined in the class Primitives as well as its subclasses refer to methods which are implemented in corresponding classes of type IMessageRecipient. In order to determine whether the Agent itself or a specific, incorporated AgentComponent is responsible for processing an incoming message with a specific primitive, the Agent as well as all incorporated AgentComponents register their primitives contained in the corresponding concrete Primitives classes with the Agent. If an Agent receives a message, it identifies which IMessageRecipient is responsible for processing the message. If there is no IMessageRecipient that registered for the message's primitive, a MessageProcessingException is thrown. The registration of Primitives with an Agent is done automatically when creating a new instance of Agent or AgentComponent (note that an Agent registers AgentPrimitives with itself). The primitives that are registered are all String values listed in an instance of the corresponding Primitives class that is retrieved by calling the previously mentioned method getPrimitives(), which has to be implemented in and adjusted by every IMessageRecipient. This automatic process makes modification of an IMessageRecipient's Primitives rather simple. Regarding AgentComponents, the automatic registration of primitives can be suppressed by using the constructor AgentComponent(T, boolean).

3.2.5 Delivery and Processing of Messages

Whenever an Agent receives a message, this message is processed within its own thread. Each Agent can thus process multiple messages *simultaneously*. Further, an Agent processes a message *in parallel to* its other actions. These two characteristics imply that it is of utmost importance that all data access triggered by messaging is implemented in a *thread-safe* way.

When an agent receives a message which is no reply but an event or a request, the agent searches the object, i.e., the target, that has been registered for this primitive. If no such target exists, a MessageProcessingException is thrown. If a target was found, the agent searches a *public* method in the target's class whose *name* equals the part of the specified primitive that identifies the method and whose *parameter is assignable from the payload*. As mentioned above, this procedure is based on using Java reflection. If such a method was found, the agent invokes this method with the message's payload. That implies that a method to be triggered by messages has to have either *zero or exactly one parameter* that implements the interface `Serializable`. If the payload is `null`, then it is expected that the method to be called does not await any parameter. Vice versa, whenever the method awaits a parameter, the payload must not be `null`. If no appropriate method could be found, a `MessageProcessingException` is thrown.

If a message is marked as a request, then the object that is returned by the invoked method is automatically sent back as a reply. The agent that awaits the reply has to cast the returned object to its concrete type. So the sender of a message has to know the type of the data with which the message's receiver responds. If a method returns `void` but the sender awaits a reply, the reply's payload is `null`. To simplify debugging, it is recommended that methods invoked by request messages do not return `void`.

Whenever an exception is thrown in the course of processing a method that was invoked by a request message, this exception is caught by the receiver, the exception's cause is put into a `ReplyMessageException`, and sent back to the sender as reply. As exceptions are forwarded to the request's sender, the sender has the possibility to react to and treat failures. When the sender receives a reply that informs about a failure, it is identified that the message contains an exception, whereupon the contained `ReplyMessageException` is thrown on the sender's side. Therefore, the sender should catch `ReplyMessageExceptions`.

If a sender does not receive a reply within the specified time, a `TimeoutException` is thrown. If the reply is received with a delay that exceeds the timeout, it is discarded. Consequently, besides catching `ReplyMessageExceptions`, it is necessary to catch `TimeoutExceptions`.

Let us take a closer look at message delivery and processing. In the following example, `MySenderAgent` is an `Agent` that wants to request information from `MyReceiverAgent` with identifier `receiverAgentIdentifier`. `MySenderAgent` chooses the corresponding primitive (`SAY_HELLO_TO_ME`) contained in `MyReceiverAgentPrimitives`. Furthermore, `MySenderAgent` needs to know that the reply sent by `MyReceiverAgent` is a `String` and the payload consists of exactly one `String` object. `MySenderAgent` calls the method `sendRequestAndWaitForReply(...)` using the right arguments. Having received the reply, it casts the returned object to `String`. In this case, the `replyString` states "Hello you!".

```java
// agent sending a message
public class MySenderAgent extends Agent {
    ...
    public void doSomething() {
        ...
        try {
            // send a request with primitive SAY_HELLO_TO_ME and payload
            // "you" to the agent with identifier receiverAgentIdentifier
            // and wait for its reply which is known to be a String
            String replyString = (String) this.sendRequestAndWaitForReply(
                receiverAgentIdentifier,
                MyReceiverAgent.getPrimitives().SAY_HELLO_TO_ME, "you",
                Agent.DEFAULT_TIMEOUT);
        }
        // catch TimeoutExceptions
        catch (TimeoutException te) {
            ...
        }
        // catch ReplyMessageExceptions
        catch (ReplyMessageException re) {
            ...
        }
        // print the reply
        System.out.println(replyString);
    }
}

// agent receiving the message
public class MyReceiverAgent extends Agent {
    ...
    // methods that are invoked by messaging need to be public
    public String sayHelloToMe(String me) {
        return "Hello " + me + "!";
    }
    ...
    // the getPrimitives() method necessary for proper messaging
    public static MyReceiverAgentPrimitives getPrimitives() {
        // MyReceiverAgentPrimitives would contain the primitive SAY_HELLO_TO_ME
        return new MyReceiverAgentPrimitives(MyReceiverAgent.class);
    }
}
```

The agent `MyReceiverAgent` has to implement the `getPrimitives()` method in order to inform others about its primitives. In this example, the primitive `SAY_HELLO_TO_ME` refers to the method `sayHelloToMe`. It is mandatory that this method is `public`. As `MySenderAgent` has sent the correct payload for the method `sayHelloToMe` (a `String` object), this method is correctly invoked and the returned `String` is automatically sent back to `MySenderAgent`.

3.2.6 MessageProcessingException

```
public class MessageProcessingException extends RuntimeException
```

This exception is thrown if an error occurs while an `Agent` processes a message it received. A message whose primitive has not been registered for a specific target, i.e., the `Agent` itself or its `AgentComponents`, or whose payload is of the wrong type so that the associated target method cannot be invoked, are typical situations in which a `MessageProcessingException` is thrown.

3.2.7 ReplyMessageException

```
public class ReplyMessageException extends RuntimeException
```

This exception is used to inform the sender of a request message that an exception was thrown while the request message was processed by the receiver. The sender throws this exception again when it receives the reply.

3.2.8 IServiceBindingProvider

```
public interface IServiceBindingProvider<T>
```

This interface is responsible for providing a service binding for a specific class that extends `T`. Service bindings play an important role for the correct delivery of messages to services (see Section 2.2) and thus `Agents`. A `Service` can only receive those messages addressing service bindings it is bound to. The `IServiceBindingProvider` is especially useful for addressing broadcast messages to specific `Services` because messages are delivered to every `Service` in the system bound to the specified service binding or a subbinding of it.

Generic Type Parameters

- **T:** The type for which service bindings can be requested.

  ```
  T extends IMessageHandlingAgent
  ```

Important Methods

- **getServiceBindingForClass:** Returns the service binding for the given class or `null` if the class is not of a valid type, i.e., if it does not extend `T`.

  ```
  public String getServiceBindingForClass(Class<? extends T> aClass);
  ```

- **getServiceBindingsForClass:** Returns a `Set` of service bindings (including subbindings) for the given class or an empty `Set` if the class is not of a valid type, i.e., if it does not extend `T`.

  ```
  public Set<String> getServiceBindingsForClass(Class<? extends T> aClass);
  ```

Usage `ServiceBindingProvider` (see Section 3.2.9) implements this interface.

3.2.9 ServiceBindingProvider

```
public class ServiceBindingProvider implements IServiceBindingProvider<Agent>
```

This class is an implementation of the interface `IServiceBindingProvider` and provides service bindings for a given type in the form of a class. Valid types are restricted to subclasses of `Agent`. The most specific service binding for a given class consists of a string representation of the reverse class hierarchy down to the class `Agent` and a project-specific suffix that prevents name resolution conflicts. Less specific subbindings are generated on the basis of the superclasses of this class. Each object of a specific class has thus the same service bindings. If we have a class hierarchy Agent → A → B → C and a project-specific suffix "myProject", then the most specific service binding of B is "B.A.Agent.myProject". subbindings are "A.Agent.myProject" and "Agent.myProject".

As in some cases, e.g., when using Repast's GUI, different system class loaders are used which possibly cannot get information about the class hierarchy of some classes, a `ClassLoader` has to be specified when a service binding is queried. This implies that exactly one subclass of `ServiceBindingProvider` is needed for each project (an example is given below). This subclass is then set in the constructor of `Agent` classes defined the corresponding project (see Section 3.1.4).

The `ServiceBindingProvider` is implemented as Singleton.

Constructors The constructor is empty and `protected` as this class is an implementation of the Singleton pattern. This class and its subclasses are thus accessed via the method `getInstance(...)`.

```
protected ServiceBindingProvider() { }
```

Important Methods

- **getInstance:** Returns a reference to the Singleton object of this class with the given project-specific `suffix` and `ClassLoader`.

  ```
  public static ServiceBindingProvider getInstance(String suffix,
      ClassLoader cl) {...}
  ```

- **getServiceBindingForClass:** Returns the most specific (cached) service binding for the given class or `null` if the class is not an `Agent`. The service binding is a `String` representing the inverse class hierarchy down to `Agent` with the `suffix` at the end (see example below). Thus, each object of the same class has the same service binding.

  ```
  public String getServiceBindingForClass(Class<? extends Agent>
      aClass) {...}
  ```

- **getServiceBindingsForClass:** Returns the (cached) service bindings (including subbindings) for the given class or an empty `Set` if the class is not an `Agent`.

  ```
  public Set<String> getServiceBindingsForClass(Class<? extends Agent>
      aClass) {...}
  ```

Usage This class should be extended in every project by a project-specific implementation of `ServiceBindingProvider` with a project-specific `suffix` and `ClassLoader`. The TEMAS project already provides such an extension, the `TEMASServiceBindingProvider`. Subclasses in other projects can be implemented according to the following example of the `TEMASServiceBindingProvider`:

```java
// a project-specific service binding provider
public class TEMASServiceBindingProvider extends ServiceBindingProvider {

    private final static ServiceBindingProvider SERVICE_BINDING_PROVIDER;

    static {
        // use project-specific parameters
        SERVICE_BINDING_PROVIDER = ServiceBindingProvider.getInstance("temas",
            TEMASServiceBindingProvider.class.getClassLoader());
    }

    // override method with project-specific service binding provider
    public static ServiceBindingProvider getInstance() {
        return SERVICE_BINDING_PROVIDER;
    }
}
```

To request the service binding of a specific `Agent`, the `ServiceBindingProvider` should be used as follows. Let `SomeAgent` be an extension of `Agent` located within the project of `TEMASServiceBindingProvider`:

```java
public class SomeAgent extends Agent {
    ...
    public SomeAgent() {
        // call super(...) with project-specific service binding provider
        super(TEMASServiceBindingProvider.getInstance(),
            TimeProviderICBased.class);
        ...
    }
    ...
}
```

Then the following call (which can be anywhere in the application, independent of the project structure) would return the most specific service binding of `SomeAgent`, namely "SomeAgent.Agent.temas":

```java
// get service binding for SomeAgent
ServiceBindingProvider.getServiceBindingForClass(SomeAgent.class);
```

3.3 Discovering Agents

Discovery is a concept orthogonal to the `YellowPages` (see Section 3.7). The TEMAS discovery functionality differs from the functionality provided by the `YellowPages` in the way that it provides a mechanism to discover agents that are bound to at least one of a set of specific service bindings, instead of providing functionality to discover agents according to their type. The TEMAS discovery functionality is based on the TEM's discovery infrastructure (see Section 2.6). Agent discovery is performed in a dedicated thread that runs a `DiscoveryTask` (see Section 3.3.1). After a certain amount of time, a `DefaultDiscoveryResult` can be retrieved from the `DiscoveryTask`.

The discovery functionality can be accessed via the `Agent`'s methods `discoverAgents`, to trigger a discovery, and `evaluateDiscoveryResult`, to process the result of the discovery (see Section 3.1.4).

3.3.1 DiscoveryTask

```
public class DiscoveryTask extends FutureTask<DefaultDiscoveryResult>
```

A `DiscoveryTask` is a `FutureTask`[6] running in its own thread. It provides a simple interface to retrieve the result of a terminated `Discovery`. This result is returned in the form of a `DefaultDiscoveryResult` when the `Agent`'s method `discoverAgents(...)` is called.

Important Methods

- **waitForDiscoveryResult:** Waits for the `Discovery` to terminate. The time to wait depends on the timeout stated when calling the method `discoverAgents(...)` at an `Agent`.

  ```
  public DefaultDiscoveryResult waitForDiscoveryResult() {...}
  ```

Usage This class is returned when the `Agent`'s method `discoverAgents(...)` is called. For a complete example how to discover `Agents`, look at the in Section 3.3.2.

3.3.2 DefaultDiscoveryResult

```
public class DefaultDiscoveryResult implements DiscoveryResult
```

This class is an implementation of OCµ's `DiscoveryResult` (see Section 2.6.3) and is therefore able to receive requested discoveries. A `DefaultDiscoveryResult` of a started `Discovery` can be obtained from the `DiscoveryTask` (see Section 3.3.1) returned by the `Agent`'s method `discoverAgents(...)`. It can then be further evaluated by calling the `Agent`'s method `evaluateDiscoveryResult(...)`.

Usage Assume that some arbitrary `Agent` wants to discover `Agents` that are registered for `SomeAgent`'s service binding as well as `Agents` that are registered for `AnotherAgent`'s service binding:

```
// get service bindings of SomeAgent and AnotherAgent
String[] bindings = new String[] { TEMASServiceBindingProvider.getInstance().
    getServiceBindingForClass(SomeAgent.class), TEMASServiceBindingProvider.
    getInstance().getServiceBindingForClass(AnotherAgent.class) };
```

Then a call of `discoverAgents(...)` returns a `DiscoveryTask` which itself returns a `DefaultDiscoveryResult` when calling its method `waitForDiscoveryResult()`. The obtained `DefaultDiscoveryResult` then can be further evaluated in the `Agent`'s method `evaluateDiscoveryResult(...)` which returns the set of `AgentIdentifiers` retrieved in the discovery:

[6] A cancellable asynchronous computation running its own thread. Information about the interface `java.util.concurrent.Future` and the class `java.util.concurrent.FutureTask` can be found in the Java documentation.

```
// start a discovery with the specified service bindings to discover and a given
// timeout of 100 ms, wait for its result, and evaluate it
Set<AgentIdentifier> ids = this.evaluateDiscoveryResult(
    this.discoverAgents(100, bindings).waitForDiscoveryResult());
```

In the example, the set of `AgentIdentifiers` contains only those agents that are bound to a service binding included in `bindings`.

3.4 Additional Trust Concepts

The TEMAS extends the TEM's Trust Metric Infrastructure (see Section 2.7) by implementing those key concepts necessary to measure and use trust in MAS (see [2] and Figure 10) that are not represented in the TEM yet. These are explicit representations of `Interactions`, `Experiences`, and `TrustContexts`. The members of these classes are derived from the associations between these concepts. As mentioned in Section 2.7, the `TrustMetric` depicted in Figure 10 is represented in the TEM by a combination of `Transformer` (see Section 2.7.3) and `Interpreter` (see Section 2.7.4). The `TrustValue` corresponds to the TEM's `TrustData`.

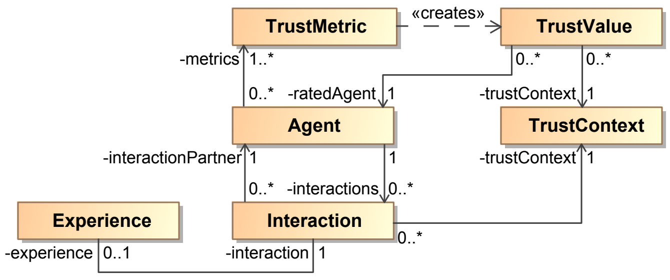

Figure 10: Concepts for the measurement and use of trust in a MAS [2]

3.4.1 Interaction

```
public abstract class Interaction<TC extends TrustContext,
    E extends Experience<?>> implements Serializable
```

This is an abstract class representing an `Interaction` that takes place between an interaction initiator and an interaction partner in a specific `TrustContext`. The outcome of this `Interaction` is stored as an `Experience`.

Generic Type Parameters

- **TC:** The concrete type of `TrustContext` in which this `Interaction` takes place.

```
TC extends TrustContext
```

- **E:** The concrete type of `Experience` which holds the outcome of this `Interaction`.

```
E extends Experience<?>
```

Constructors Creates a new `Interaction` that took place in a specific `TrustContext` between an `interactionInitiator` and an `interactionPartner`. The given `Strings` `interactionInitiator` and `interactionPartner` both represent the `AgentIdentifier` of the corresponding `Agents` (see Section 3.1.5).

```
public Interaction(TC trustContext, String interactionInitiator,
    String interactionPartner)
```

Important Members

- **experience:** The `Experience` that was gained **after** evaluating this `Interaction`. This `experience` is `null` if the `Interaction` has not been evaluated yet.

  ```
  private E experience;
  ```

- **interactionInitiator:** The initiator of this `Interaction` in the form of a string representation of the corresponding `AgentIdentifier`.

  ```
  private final String interactionInitiator;
  ```

- **interactionPartner:** The partner in this `Interaction` in the form of a string representation of the corresponding `AgentIdentifier`.

  ```
  private final String interactionPartner;
  ```

- **trustContext:** The `TrustContext` this `Interaction` was made in.

  ```
  protected final TC trustContext;
  ```

Important Methods

- **getExperience:** Returns the associated `Experience`.

  ```
  public E getExperience() {...}
  ```

- **setExperience:** Sets the associated `Experience`. Having set an `Experience`, it cannot be replaced afterwards. Returns `true` if and only if the `Experience` has been associated with the `Interaction`.

  ```
  public boolean setExperience(E experience) {...}
  ```

- **hasExperience:** Returns `true` if this `Interaction` has an associated `Experience`, `false` if not.

  ```
  public boolean hasExperience() {...}
  ```

- **getInteractionInitiator:** Returns the `interactionInitiator`, i.e., its identifier.

```
public String getInteractionInitiator() {...}
```

- **getInteractionPartner:** Returns the `interactionPartner`, i.e., its identifier.

  ```
  public String getInteractionPartner() {...}
  ```

- **getTrustContext:** Returns the `TrustContext` this `Interaction` was made in.

  ```
  public TC getTrustContext() {...}
  ```

Usage This abstract class needs to be extended by a concrete `Interaction`.

3.4.2 Experience

```
public abstract class Experience<I extends Interaction<?, ?>>
    implements Serializable
```

This is an abstract class representing an `Experience` that was made in the course of an `Interaction` with a specific interaction partner.

Generic Type Parameters

- **I:** The concrete type of `Interaction` associated with this `Experience`.

  ```
  I extends Interaction<?, ?>
  ```

Constructors Creates a new `Experience` for a specific `Interaction` and associates the `Experience` with the `Interaction`. If the given `Interaction` already has an `Experience`, an `IllegalArgumentException` is thrown. Note that the `Interaction` has also be associated with the `Experience` by calling `setExperience(...)`.

```
public Experience(I interaction)
```

Important Members

- **interaction:** The `Interaction` for which the `Experience` was made.

  ```
  private final I interaction;
  ```

Important Methods

- **getInteraction:** Gets the `Interaction` associated with this `Experience`.

  ```
  public I getInteraction() {...}
  ```

Usage This abstract class needs to be extended by a concrete `Experience`.

3.4.3 TrustContext

```
public abstract class TrustContext
```

Instances of this abstract class represent contexts in which `Interactions` take place and `Experiences` are gained. The representation of different trust contexts by different classes allows to establish relationships between different trust contexts. In the Trust Metric Infrastructure of the TEM (see Section 2.7), the `TrustContext` is stored as a `String`. Therefore, every `TrustContext` needs a `String` representation which returns an equal `String` for `Interactions` made in the same `TrustContext`. We decided to represent trust context by the class `TrustContext` instead of a string because this allows to be able to properly describe the relations between different trust contexts.

Important Methods

- **getStringRepresentation:** The default implementation of this method returns the Java fully qualified class name of this `TrustContext`.

  In case subclasses override this method, it is essential that this method returns an equal `String` for `TrustContext` objects that are of the same type.

  ```
  public String getStringRepresentation() {
      return this.getClass().getName();
  }
  ```

Usage This abstract class needs to be extended by a concrete `TrustContext`. In case the default implementation of `getStringRepresentation()` is not sufficient, this method has to be overridden.

3.4.4 Trust-Based Scenarios

In this section, we briefly introduce the most important concepts used to generate *Trust-Based Scenarios* (TBS) [1]. TBS are an instrument to predict an agent's, a subsystem's, or the environment's future behavior by approximating the underlying stochastic process. Each TBS gives information about a specific future development and a probability of occurrence.

Because TBS can be used to make predictions about multiple future time steps, experiences are not only gained once at the end of an interaction. Instead, in the course of an interaction, an experience is gained in and for each single time step. Interactions and experiences are thus separated into distinct concepts called `AtomicInteraction` and `InteractionContainer`, and `AtomicExperience` and `ExperienceContainer` (see Figure 11). All these concepts are subclasses of `Interaction` (see Section 3.4.1) or `Experience` (see Section 3.4.2). `AtomicInteractions` or `AtomicExperiences` are interactions or experiences that take place or are gained in a single time step. The

InteractionContainer or ExperienceContainer contain all AtomicInteractions or AtomicExperiences that belong to a specific interaction that takes place over multiple time steps.

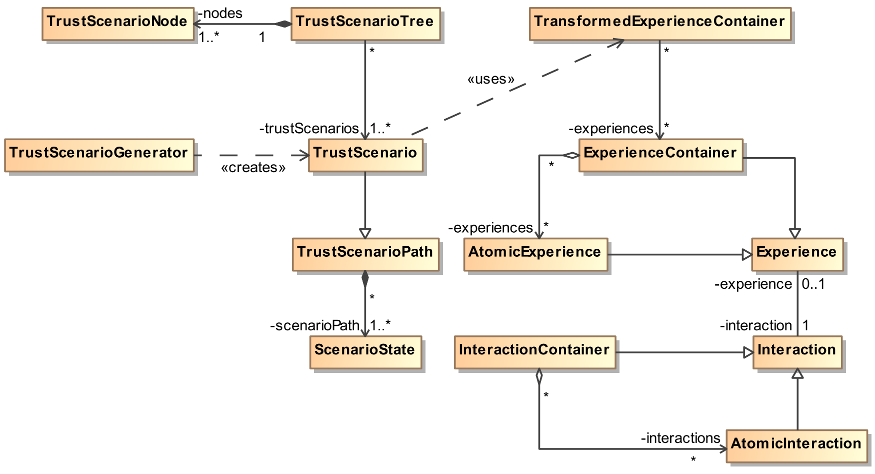

Figure 11: Trust-Based Scenarios

By using a trust metric consisting of a transformer and interpreter as shown in Section 2.7, ExperienceContainers previously stored in the TEM are transformed into a TransformedExperienceContainer, a subclass of TransformedData, which basically serves as a wrapper for the set of ExperienceContainers that are to be interpreted by the interpreter. Here, the interpreter instantiates a new TrustScenarioGenerator, a subclass of TrustData, with the TransformedExperienceContainer as parameter. The TrustScenarioGenerator provides an interface to generate TBS by evaluating the given TransformedExperienceContainer. The TBS are represented by the class TrustScenario. A TrustScenario is a TrustScenarioPath with a probability of occurrence. A TrustScenarioPath is a list of ScenarioStates. Each ScenarioState belongs to a specific future time step that depends on the position in the list. A ScenarioState is thus a prediction of the rating of an agent's, a subsystem's, or the environment's future behavior in a specific TBS and a specific time step. However, the ScenarioState does not predict the rating of the future behavior in the form of a single value but by a range of expected values described by an upper and lower threshold, i.e., an interval. Consequently, a TrustScenario represents a corridor of expected behavior.

Several TrustScenarios can be represented by a TrustScenarioTree that consists of TrustScenarioNodes, each representing a particular ScenarioState.

3.5 Extending Agent Concepts by the Notion of Trust

On the basis of the TEM's Trust Metric Infrastructure (see Section 2.7) and the trust concepts defined by the TEMAS, the concept Agent can be extended to a TrustAgent, i.e., an agent that is able to gain Experiences in Interactions and derive trust values and other trust-related data from this information. While the interface `ITrustAgent` defines the general capabilities a trust-aware agent should have, the `TrustAgent` extends the class `Agent` and provides a concrete implementation of the `ITrustAgent` interface.

3.5.1 ITrustAgent

```
public interface ITrustAgent extends IMessageHandlingAgent
```

An `ITrustAgent` is an extension of the `IMessageHandlingAgent` and is an interface that should be implemented by any class that represents an agent that is able to retrieve trust values and trust-based scenarios from the TEM.

Important Methods

- **setDirectTrustMetric**: Sets a direct trust metric, given in the form of a combination of `Transformer` and `Interpreter`, for the combination of `targetAgentType`, `Facet`, and `TrustContext`. The trust metric can be used later on to evaluate collected `Experiences`. Details depend on the concrete implementation.

```
public <R extends RawData, T extends TransformedData, U extends TrustData>
    void setDirectTrustMetric(String targetAgentType, Facet facet,
    TrustContext context, Transformer<R, T> transformer,
    Interpreter<T, U> interpreter);
```

- **addDirectTrustRawData**: This method adds `RawData` that contain a list of `Experiences` this `ITrustAgent` gained in `Interactions` with another `Agent` (identified by the given `targetAgentIdentifier`) in the specified `TrustContext` and for the given `Facet` in a data base containing the `ITrustAgent`'s `Experiences`. Details depend on the concrete implementation.

```
public void addDirectTrustRawData(String targetAgentIdentifier,
    Facet facet, TrustContext trustContext, RawData rawData);
```

- **getDirectTrustValue:** This method retrieves `TrustData` describing the trustworthiness of an `Agent` (identified by the given `targetAgentIdentifier`) regarding a specific `Facet` and `TrustContext`. The `TrustData` is based on `Experiences` gained by this `ITrustAgent`. Details depend on the concrete implementation.

```
public TrustData getDirectTrustValue(String targetAgentIdentifier,
    Facet facet, TrustContext trustContext);
```

- **getTrustBasedScenarios:** This method retrieves an `ITrustScenarioGenerator` which can be used to generate trust-based scenarios for a specific `Agent` (identified by the given `targetAgentIdentifier`), `Facet`, and `TrustContext`. Details depend on the concrete implementation.

```
public ITrustScenarioGenerator<?> getTrustBasedScenarios(
    String targetAgentIdentifier, Facet facet, TrustContext trustContext);
```

Usage This interface is implemented by the class `TrustAgent`.

Note Please note that the `ITrustAgent` and its subclasses currently do **not** feature an interface for obtaining reputation values provided by the TEM.

3.5.2 TrustAgent

```
public abstract class TrustAgent extends Agent implements ITrustAgent
```

A `TrustAgent` is an `Agent` that implements the `ITrustAgent` interface and thus provides additional methods which allow the `TrustAgent` and assigned `AgentComponents` to store `Experiences` in the form of `RawData` in the TEM and derive trust values and trust-based scenarios by using individual trust metrics.

Constructors Creates a new instance of `TrustAgent` (see also Section 3.1.4).

```
public TrustAgent(ServiceBindingProvider serviceBindingProvider,
    Class<? extends DefaultTimeProvider> timeProviderClass) {...}
```

Important Methods

- **setDirectTrustMetric**: This method is defined by the `ITrustAgent` interface. It sets the direct trust metric that is used to evaluate collected `Experiences` previously stored in the TEM and is based on the method `setTrustMetric(...)` defined in the interface `Trust` of the TEM. The specified `targetAgentType` can either be a service type (as given by `Agent.getServiceType()`), a service id (as given by `Agent.getServiceId()`), or a `String` representation of the agent identifier (as given by `Agent.getAgentIdentifier()`). No matter which information is actually given, the service type is extracted and the trust metric, given in the form of a `Transformer` and `Interpreter`, is registered for the given combination of service type, `Facet`, and `TrustContext`.

```
public <R extends RawData, T extends TransformedData, U extends TrustData>
    void setDirectTrustMetric(String targetAgentType, Facet facet,
    TrustContext context, Transformer<R, T> transformer,
    Interpreter<T, U> interpreter) {...}
```

- **addDirectTrustRawData**: This method is defined by the `ITrustAgent` interface. The method adds `RawData` that contain `Experiences` this `TrustAgent` gained in `Interactions` with an `Agent` represented by the given `targetAgentIdentifier` in the specified `TrustContext` and for the given `Facet` in the TEM.

```
public void addDirectTrustRawData(String targetAgentIdentifier,
    Facet facet, TrustContext trustContext, RawData rawData) {...}
```

- **getDirectTrustValue**: This method is defined by the `ITrustAgent` interface. The data used to calculate the trust value are `Experience`s previously stored in the TEM by calling `addDirectTrustRawData(...)`. The trust value is derived by means of a trust metric set by calling `setDirectTrustMetric(...)`.

```
public TrustData getDirectTrustValue(String targetAgentIdentifier,
    Facet facet, TrustContext trustContext) {...}
```

- **getTrustBasedScenarios**: This method is defined by the `ITrustAgent` interface. As is the case with `getDirectTrustValue(...)`, the `ITrustScenarioGenerator` returned by this method is based on `Experience`s stored in the TEM.

```
public ITrustScenarioGenerator<?> getTrustBasedScenarios(String
    targetAgentIdentifier, Facet facet, TrustContext trustContext) {...}
```

Usage This class should be extended by every agent in the system that is able to use trust concepts.

3.6 Time Concepts

In this section, we present interfaces and classes in the TEMAS that enable agents to use basic time concepts.

3.6.1 ITimeProvider

```
public interface ITimeProvider
```

This interface provides a method to request the current time. This information is returned in form of a `GregorianCalendar`.

Important Methods

- **getCurrentTime:** Returns the current time in form of a `GregorianCalendar`.

```
public GregorianCalendar getCurrentTime();
```

- **initTimeProvider:** Initializes this `ITimeProvider`.

```
public void initTimeProvider();
```

Usage This interface is extended by the interface `IAgent` (as every agent should be able to request the current time) and implemented by the class `DefaultTimeProvider` and its extension, the `TimeProviderICBased` (see Section 3.6.3).

3.6.2 DefaultTimeProvider

```
public class DefaultTimeProvider extends AgentComponent<IMessageHandlingAgent>
    implements ITimeProvider
```

This class is an `AgentComponent` and an implementation of the interface `ITimeProvider`. `Agent` can use the `DefaultTimeProvider` to request the current local system time. This class should be extended by all application-specific `ITimeProvider` implementations that are to be used by an `Agent`.

Constructors Creates a new instance of the `DefaultTimeProvider`. As this class is a subclass of `AgentComponent`, the constructor of `AgentComponent` is called.

```
public DefaultTimeProvider(IMessageHandlingAgent owner) {...}
```

Important Methods

- **getCurrentTime:** Returns the current local system time as returned by `new GregorianCalendar()`.

  ```
  public GregorianCalendar getCurrentTime() {...}
  ```

- **initTimeProvider:** This method is empty as nothing has to be initialized.

  ```
  public void initTimeProvider() { }
  ```

Usage This class is extended by the `TimeProviderICBased`.

3.6.3 TimeProviderICBased

```
public class TimeProviderICBased extends DefaultTimeProvider
```

This extension of the `DefaultTimeProvider` and implementation of the `ITimeProvider` gets the current time from a unique `Agent`, called the `InternalCalendar`, that provides the global system time (see Section 3.6.4). The time is consequently communicated via messages. For performance reasons and less message overhead, the `InternalCalendar` sends, whenever the current time is updated, the current time to all registered instances of `TimeProviderICBased`. The instances of `TimeProviderICBased` that receive this message then cache the updated time. As long as the time is not updated, an instance of `TimeProviderICBased` provides its owner with the cached time.

Constructors Creates a new instance of the `TimeProviderICBased`. As this class is a subclass of `AgentComponent`, the constructor of `AgentComponent` is called.

```
public TimeProviderICBased(IMessageHandlingAgent owner) {...}
```

Important Members

- **cachedCurrentTime:** Holds the current time in form of a `GregorianCalendar` that is sent by the `InternalCalendar` whenever the time changes.

  ```
  private GregorianCalendar cachedCurrentTime;
  ```

Important Methods

- **getCurrentTime:** Returns the current time that is stored in `cachedCurrentTime`, or, if no `cachedCurrentTime` is set, the `TimeProviderICBased` requests the current time from the `InternalCalendar`.

  ```
  public GregorianCalendar getCurrentTime() {...}
  ```

- **initTimeProvider:** Registers this `AgentComponent` for time updates sent by the system-wide unique `InternalCalendar`. From then on, this `AgentComponent` is informed about the current time by the `InternalCalendar` every time the time changes. This method has to be called once. This method is automatically called when an `Agent` that uses the `TimeProviderICBased` as `DefaulTimeProvider` is initialized.

  ```
  public void initTimeProvider() {...}
  ```

- **informCurrentTime:** Updates `cachedCurrentTime` with the given `currentTime`. The method is called when the `InternalCalendar` sends a message with an updated current time.

  ```
  public void informCurrentTime(GregorianCalendar currentTime) {...}
  ```

Usage `Agents` that want to use this implementation of `DefaultTimeProvider` have to specify this class when calling the `Agent`'s constructor.

3.6.4 InternalCalendar

```
public class InternalCalendar extends Agent
```

This `Agent` represents a clock that provides a global system time to all `Agents`. Therefore, there must not be more than one `InternalCalendar` in the entire system. The `InternalCalendar` holds the current time in form of a `GregorianCalendar`. In contrast to the `DefaultTimeProvider` (see Section 3.6.2), the `InternalCalendar` measures the time in discrete time steps called *ticks*. Each time the `InternalCalendar` executes its `achieveGoals()` method, a tick has passed and the current time hold in the `InternalCalendar` is updated accordingly. More precisely, the current time is updated by incrementing the prior current time by an application-specific number of seconds. If desired, the `InternalCalendar` is able to skip a predefined time interval, e.g., if the simulation should skip a specific time frame, e.g., the night. Before using the `InternalCalendar`, it has to be created and initialized by calling the `initialize(...)` method. Otherwise, if the `InternalCalendar` has not been initialized, it does not exist and an `InternalCalendarNotInitializedException` is thrown. For performance reasons and less message overhead, the `InternalCalendar` sends, whenever the current time is updated, the current time to all registered instances of `TimeProviderICBased`.

The `InternalCalendar` is implemented according to the singleton pattern. However, this does not prevent the creation of multiple instances in a distributed system. This is why the method `initialize(...)` must be called multiple times on different nodes.

Constructors The private constructor creates a new instance of `InternalCalendar` and is called when calling the method `initialize(...)`. The `InternalCalendar`'s time `theCalendar` is set to `simulationStartTime`, thus defining the simulation start time. The given `secondsPerTick` defines the amount of time `theCalendar` is incremented by each tick. If `avoidTimeInterval` is `true`, the time interval specified by the given `avoidTimeIntervalStart` (the hour of the start date to be skipped) and the given `avoidTimeIntervalDuration` (the number of hours to skip) is skipped when incrementing the time.

```
private InternalCalendar(GregorianCalendar simulationStartTime,
    int secondsPerTick, boolean avoidTimeInterval,
    int avoidTimeIntervalStart, int avoidTimeIntervalDuration) {...}
```

Important Members

- **theCalendar:** Holds the current time in form of a `GregorianCalendar` that is incremented by `secondsPerTick` (except if the time interval to skip is reached) each tick.

```
private GregorianCalendar theCalendar;
```

- **currentTick:** Holds the current tick of the simulation.

```
private long currentTick;
```

- **secondsPerTick:** Holds the seconds `theCalendar` is incremented by each tick.

```
private static int secondsPerTick;
```

- **registeredTimeProviders:** Holds identifiers of registered instances of the class `TimeProviderICBased` that are informed about current time updates.

```
private HashSet<String> registeredTimeProviders;
```

Important Methods

- **initialize:** This method needs to be called before accessing the `InternalCalendar` in order to instantiate and initialize it. The method calls the private constructor of `InternalCalendar` with a given simulation start time `simulationStartTime`, the `secondsPerTick` the time is incremented by in each tick, and whether or not a specific time interval should be skipped.

```
public static void initialize(GregorianCalendar simulationStartTime,
    int secondsPerTick, boolean avoidTimeInterval,
    int avoidTimeIntervalStart, int avoidTimeIntervalDuration) {...}
```

- **getInstance:** Returns the singleton instance of `InternalCalendar`. However, before first calling this method, the `InternalCalendar` has to be initialized, else an `InternalCalendarNotInitializedException` is thrown.

```
public static synchronized InternalCalendar getInstance() {...}
```

- **registerForCurrentTimeUpdates:** Adds the given `timeProviderId`, i.e., the string representation of an `TimeProviderICBased`'s owner's `AgentIdentifier`, to `registeredTimeProviders`. From then on, the registered `TimeProviderICBased` does not need to request the current time manually by calling `getCalendar()` but is informed every time the `InternalCalendar` updates the global system time. The `InternalCalendar` calls this method when it receives a message with primitive "registerForCurrentTimeUpdates".

```
public void registerForCurrentTimeUpdates(String timeProviderId) {...}
```

- **getCalendar**: Returns the current time. However, the current time should not be requested by a method call but by a message with primitive "getCalendar".

```
public GregorianCalendar getCalendar() {...}
```

- **getCurrentTick:** Returns the number of ticks since the start of the simulation.

```
public long getCurrentTick() {...}
```

- **getSecondsPerTick:** Returns the seconds that `theCalendar` is incremented by in each tick.

```
public static int getSecondsPerTick() {...}
```

- **ceilToNextTick:** Returns the `Date` for the next tick in the scheme of discrete time steps after the given `date` or `date` itself if it corresponds to a `Date` that fits into the scheme.

```
public static Date ceilToNextTick(Date date) {...}
```

- **tickAfter:** Returns the `Date` for the next tick after `date`. Again, the returned `Date` fits into the scheme of discrete time steps.
  ```
  public static Date tickAfter(Date date) {...}
  ```

Usage Agents that want to request the current time from the `InternalCalendar` should set the `TimeProviderICBased` (see Section 3.6.3) as their `DefaultTimeProvider`, which uses the `InternalCalendar` to request and cache the current time.

3.6.5 InternalCalendarNotInitializedException

```
public class InternalCalendarNotInitializedException extends RuntimeException
```

This extension of `RuntimeException` is thrown when the `InternalCalendar` is tried to be accessed without having been initialized before.

3.6.6 TimeInterval

```
public class TimeInterval implements Serializable, Iterable<Date>
```

This class defines a time interval of discrete points in time with a specified start date and end date. The discretization between the start and end date is given by the `InternalCalendar`'s scheme of discrete time steps, called ticks (see Section 3.6.4). The time interval is iterable.

Constructors

- Creates a new instance of `TimeInterval` with a given `startDate` and `endDate`.
  ```
  public TimeInterval(Date startDate, Date endDate) {...}
  ```

- If only one `date` is given, a new instance of `TimeInterval` comprising only one point in time, i.e., the start and end date are the same, is created.
  ```
  public TimeInterval(Date date) {...}
  ```

- This class also provides a constructor for creating a so-called empty `TimeInterval` (with no point in time included). In this case, start and end date are set to `null`.
  ```
  public TimeInterval() {...}
  ```

Important Members

- **startDate:** Holds the interval's start date.
  ```
  private final Date startDate;
  ```

- **endDate:** Holds the interval's end date.
  ```
  private final Date endDate;
  ```

Important Methods

- **getStartDate:** Returns the `startDate` of this `TimeInterval`.

  ```
  public Date getStartDate() {...}
  ```

- **getEndDate:** Returns the `endDate` of this `TimeInterval`.

  ```
  public Date getEndDate() {...}
  ```

- **containsDate:** Returns `true` if the given `date` is contained in this `TimeInterval`.

  ```
  public boolean containsDate(Date date) {...}
  ```

- **containsTimeInterval:** Checks whether `otherInterval` is fully contained in this `TimeInterval`. If `otherInterval` is an empty interval, it is contained in this `TimeInterval`. Otherwise, `otherInterval` is contained in this interval if and only if its start date is not before this interval's start date and its end date is not after this interval's end date.

  ```
  public boolean containsTimeInterval(TimeInterval otherInterval) {...}
  ```

- **overlaps:** Returns `true` if the given `TimeInterval otherInterval` overlaps with this `TimeInterval`.

  ```
  public boolean overlaps(TimeInterval otherInterval) {...}
  ```

- **getLength:** Returns the number of discrete points in time between the `startDate` (included) and `endDate` (included).

  ```
  public int getLength() {...}
  ```

- **get:** Returns the i-th date within this `TimeInterval`, e.g., `get(0)` returns the `startDate`.

  ```
  public Date get(int i) {...}
  ```

3.7 YellowPages

Sometimes it is necessary for a system participant to find out which agents of a specific type exist in the system. This information can often be requested from a central knowledge source similar to the well known yellow pages. The MASConcepts4TEM provides an `Agent` that is responsible for this functionality, the `YellowPages`. Despite the `YellowPages` is a central facility and thus might seem to be a source of single point of failure, in future work, the TEM provides mechanisms to identify situations in which mandatory services are unavailable and restart such services on more reliable nodes. Please note that the `YellowPages` are an orthogonal concept to the discovery functionality of OCµ (see Section 2.6). While the discovery functionality allows to identify `Agents` that are bound to specific service bindings, the `YellowPages` allows to identify `Agents` that belong to a specific type.

3.7.1 YellowPages

```
public class YellowPages extends Agent
```

The `YellowPages` is an `Agent` which provides information about which `Agents` in the system are of a specific requested type, e.g., which agents implement the interface `ITrustAgent`. New `Agents` are automatically registered with the `YellowPages` by sending `YellowPagesData` (see Section 3.7.2) to the `YellowPages` when the `Agent`'s method `initAgent(...)` is called.

Constructors Creates a new `YellowPages` `Agent`.

```
public YellowPages() {...}
```

Important Members

- **registeredAgentData:** Holds all `YellowPagesData` of `Agents` registered with the `YellowPages`.

  ```
  private Set<YellowPagesData> registeredAgentData;
  ```

Important Methods

- **getAgentIDsOfType:** Returns the `AgentIdentifiers` of `Agents` that are registered with the `YellowPages` and are of the type `clazz`.

  ```
  public Set<AgentIdentifier> getAgentIDsOfType(Class<? extends IAgent>
      clazz) {...}
  ```

- **registerAgentData:** Dependent on the given argument, this method registers either a `Collection` of `YellowPagesData` ...

  ```
  public void registerAgentData(Collection<YellowPagesData>
      agentDataToRegister) {...}
  ```

  ...or a single `YellowPagesData` with the `YellowPages`.

  ```
  public void registerAgentData(YellowPagesData agentDataToRegister) {...}
  ```

- **unregisterAgentData:** Removes an `Agent`'s `YellowPagesData` from the registered `YellowPagesData`, i.e., unregisters the `Agent`.

  ```
  public void unregisterAgentData(YellowPagesData agentDataToUnregister)
      {...}
  ```

- **getYellowPagesDataForAgentId:** Returns `YellowPagesData` that belong to `Agents` that are identified by the given `AgentIdentifiers`. Returns an empty list if there is no `Agent` registered with the `YellowPages` that can be identified with the given `AgentIdentifiers`.

```
public List<YellowPagesData> getYellowPagesDataForAgentId(
    List<AgentIdentifier> agentIds) {...}
```

- **getYellowPagesDataForServiceId:** Gets the `YellowPagesData` that belong to `Agents` that are identified by the given service ids. Returns an empty list if there is no `Agent` registered with the `YellowPages` that can be identified with the given service ids.

```
public List<YellowPagesData> getYellowPagesDataForServiceId(List<String>
    serviceIds) {...}
```

Usage Registering and unregistering `Agents` as well as requesting other information from the `YellowPages` should be handled via messages and the corresponding primitives of the class `YellowPages`.

3.7.2 YellowPagesData

```
public class YellowPagesData implements Serializable
```

This class represents data of an `Agent` which is used to register the `Agent` with the `YellowPages`.

Constructors Creates new data used to register with the `YellowPages`.

```
public YellowPagesData(Class<?> theClazz, AgentIdentifier theAgentIdentifier)
    {...}
```

Important Members

- **classOfRepresentedAgent:** Holds the class of the `Agent` that registers with the `YellowPages`.

```
private final Class<?> classOfRepresentedAgent;
```

- **identifierOfRepresentedAgent:** Holds the `Agent`'s `AgentIdentifier`.

```
private final AgentIdentifier identifierOfRepresentedAgent;
```

Important Methods

- **getClazz:** Gets the class of the `Agent` this `YellowPagesData` belongs to.

```
public Class<?> getClazz() {...}
```

- **getAgentIdentifier:** Gets the `Agent`'s `AgentIdentifier` this `YellowPagesData` belongs to.

```
public AgentIdentifier getAgentIdentifier() {...}
```

Usage When an `Agent`'s `initAgent(...)` method is called, the `Agent` automatically creates an instance of `YellowPagesData` and registers with the `YellowPages`.

4 Deploying TEMAS-based Applications

The TEMAS is a combination of the TEM and additional concepts and functionality consolidated in the MASConcepts4TEM (see Figure 12). While it is possible to deploy your applications on the basis of the TEM in a local or distributed runtime environment called *TEM execution environment*, you might be interested in testing your applications in a runtime environment, called *temLight simulation environment*, that is deployed locally and totally abstracts from infrastructural considerations like nodes. In the following, we present these two runtime environments in detail (see Section 4.1), explain how to initialize `Agents` dependent on the runtime environment you want to use (see Section 4.2), introduce the different execution models available in the runtime environments (see Section 4.3), give an overview of agent scheduling in the TEMAS depending on the used runtime environment (see Section 4.4), explain how to bootstrap your applications (see Section 4.5), show how to configure the TEM (see Section 4.6), and give some information on how to set up an Eclipse project when using the TEMAS (see Section 4.7).

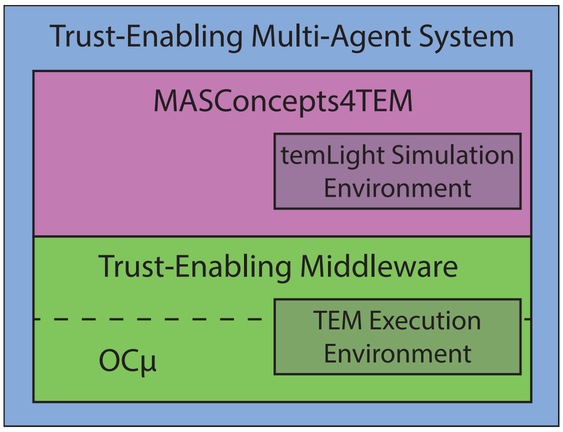
Trust-Enabling Multi-Agent System
MASConcepts4TEM
temLight Simulation Environment
Trust-Enabling Middleware
TEM Execution Environment
OCμ

Figure 12: The TEMAS architecture

4.1 TEM and temLight: Two Possible Ways to Run Your Applications

As mentioned above, there are two possible ways to run a TEMAS-based application: The first possibility is to run the application on the basis of the execution environment provided by an instance of the TEM, called *TEM execution environment*. Such an environment consists of `OcmNodes`, `Services`, and `Agents`. This instance can be locally or distributively executed so that all `Agents`, `Services`, and `OcmNodes` are hosted either on a single machine or multiple computers. In such a *local* or *distributed TEM execution environment*, at least a single `OcmNode` must be run per computer. Basic mechanisms like agent scheduling, the Trust Metric Infrastructure, message transport, and discovery of `Services` and `Agents` on the basis of service bindings are realized by the TEM (see Table 1 and Section 2.6). The TEM's message transport serializes a message's payload on the sender's side and deserializes it afterwards on the recipient's side.

The second possibility is to run the application without a TEM instance in the *temLight simulation environment* (see Table 2). Although there is no actively running instance of the TEM and thus no `OcmNodes`, the temLight simulation environment provides the TEM's Trust Metric Infrastructure. Agent scheduling, discovery on the basis of service bindings, and message transport is, however, realized by the temLight simulation environment instead of the TEM execution environment. Regarding the temLight's message transport, the object references are sent from one agent to another.

In the TEM execution environment as well as the temLight simulation environment, the discovery of `Services` and `Agents` is implemented in an asynchronous fashion. Note that the temLight simulation environment always discovers all agents that are bound to at least one of the specified service bindings, while the TEM execution environment might only discover a subset of these agents in case a timeout occurs.

	Scheduling of Agents	Trust Metric Infrastructure	Message Transport	Agent Discovery	Nodes
temLight					YES
TEM	×	×	×	×	

Table 1: TEM execution environment: responsibilities

	Scheduling of Agents	Trust Metric Infrastructure	Message Transport	Agent Discovery	Nodes
temLight	×		×	×	NO
TEM		×			

Table 2: temLight simulation environment: responsibilities

Figure 13 summarizes the most important interfaces defined by the TEM and the MASConcepts4TEM. Here, the term "interface" denotes generic architectures or functionality that is provided by the TEMAS and can be used by TEMAS-based applications out of the box. Interfaces highlighted in blue are interfaces of the TEM that are redefined by interfaces provided by the MASConcepts4TEM, meaning that the interfaces are adjusted to the needs of a MAS. In such cases, it is recommended to use the refined interfaces. Interfaces highlighted in green indicate those that provide functionality that can be easily used by your applications. The TEM basically defines five major interfaces: the `TrustServiceConnector`, the `ServiceConnector`, `Discovery` functionality, the `FailureDetectorService`, and the `TrustMetricInfrastructure`. Apart from the `FailureDetectorService` and the basic architecture for trust metrics provided by the `TrustMetricInfrastructure`, the MASConcepts4TEM use and redefine all of these interfaces. So if you want to detect malfunctioning nodes and services you have to directly use the `FailureDetectorService`, and if you want to specify your own trust metrics you have to do this according to the architecture defined by the `TrustMetricInfrastructure`. The `Agent` uses and redefines the TEM's interfaces `ServiceConnector` (used to send and receive messages) and `Discovery` (used to discover other `Agents` and `Services` that are bound to specific service bindings). The `Agent` simplifies access to this functionality as shown in Section 3.1.4. The discovery functionality is realized separately in the TEM execution environment and the temLight simulation environment. The `TrustAgent` extends the interface provided by the `Agent` by the functionality to configure trust metrics, gather experiences, and calculate trust values as well as trust-based scenarios. This functionality is based on functionality provided by the `TrustServiceConnector` and the generic architecture for trust management provided by the `TrustMetricInfrastructure`.

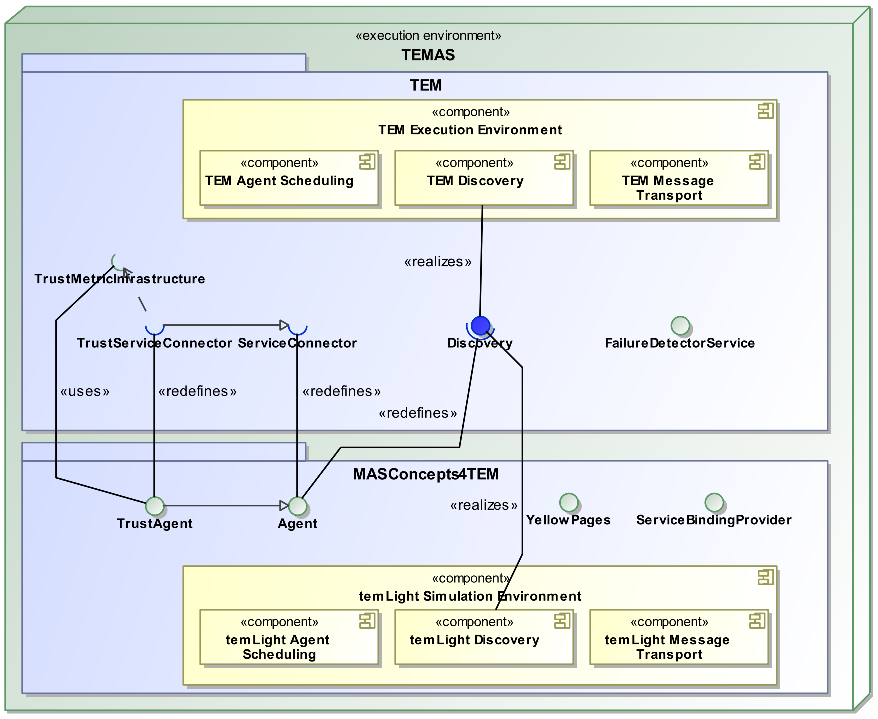

Figure 13: Relevant interfaces of the TEMAS defined by the TEM and the MASConcepts4TEM

4.2 Initializing Agents

The most important concepts for agent initialization are depicted in Figure 14. Independently of whether the TEM execution environment or temLight simulation environment is used, `Agents` have to be initialized using the `AgentInitializer` (see Section 4.2.1). In order to take the characteristics of the underlying runtime environment into account, the `AgentInitializer` makes use of an `IConcreteAgentInitializer`. The TEMAS provides two `IConcreteAgentInitializers` out of the box, namely the

TEMAgentInitializer (see Section 4.2.3) and the TemLightAgentInitializer (see Section 4.2.4). In case of the TEM execution environment, the AgentInitializer has to be initialized with the TEMAgentInitializer. If the temLight simulation environment is used, the AgentInitializer must be initialized with the TemLightAgentInitializer. The TEMAgentInitializer automatically starts new OcmNodes and registers Agents with them, whereas the TemLightAgentInitializer initializes agents independently of underlying OcmNodes because these do not exist in the temLight simulation environment. Just like the TEMAgentInitializer, the TemLightAgentInitializer ensures that each TrustAgent can make use of the TEM's Trust Metric Infrastructure.

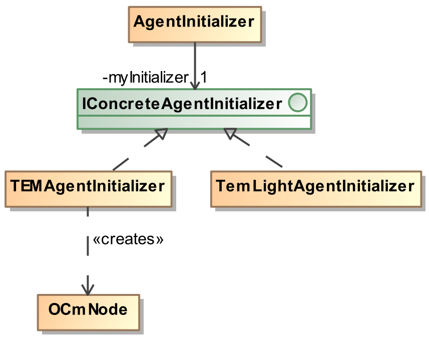

Figure 14: Concepts for initializing agents

4.2.1 AgentInitializer

```
public abstract class AgentInitializer
```

The AgentInitializer is the class that is responsible for initializing Agents. In order to initialize Agents, it uses an implementation of the interface IConcreteAgentInitializer (see Section 4.2.2). This is necessary since Agents that run in the TEM execution environment have to be initialized differently than Agents that run in the temLight simulation environment. Initializing Agents using the AgentInitializer is mandatory in order to be able to send and receive messages and use the trust concepts provided by the TEM.

Important Methods

- **initialize:** Initializes the AgentInitializer with an IConcreteAgentInitializer. The IConcreteAgentInitializer is then used to initialize all Agents in the system.

  ```
  public static void initialize(IConcreteAgentInitializer ai) {...}
  ```

- **initializeAgent:** Initializes the given `agent` using the implementation of the specified `IConcreteAgentInitializer`. `initData` is used to initialize `agent` and might be `null` if no such data is required.

  ```
  public static void initializeAgent(Agent agent,
      Map<String, Serializable> initData) {...}
  ```

Usage Before use, the `AgentInitializer` has to be initialized once when starting the system with an implementation of the `IConcreteAgentInitializer` that is to be used to initialize agents:

```
// in this example, we use the TEMAgentInitializer
AgentInitializer.initialize(new TEMAgentInitializer());
```

In this example, the agents are deployed in the TEM execution environment so that the `TEMAgentInitializer` is used. From then on, every `Agent` implementation has to be initialized as follows:

```
// create new agent
SomeAgent a = new SomeAgent();
// initialize the agent afterwards (do not forget that!)
AgentInitializer.initializeAgent(a, null);
```

Note that initializing agents after creation is mandatory.

4.2.2 IConcreteAgentInitializer

```
public interface IConcreteAgentInitializer
```

This interface provides a method to actually initialize an `Agent`.

Important Methods

- **initializeAgent:** Initializes the given `agent` with `initData`. `initData` might be `null` if no such data is required.

  ```
  public void initializeAgent(Agent agent, Map<String, Serializable> initData);
  ```

Usage This interface is extended by the `TemLightAgentInitializer` (see Section 4.2.4) and the `TEMAgentInitializer` (see Section 4.2.3).

4.2.3 TEMAgentInitializer

```
public class TEMAgentInitializer implements IConcreteAgentInitializer
```

This class is an implementation of the interface `IConcreteAgentInitializer` (see Section 4.2.2). It can be used to initialize `Agents` in a **local** TEM execution environment. `Agents` are registered and started on automatically created `OcmNodes` (see Section 2.1).

Constructors Creates a new instance of `TEMAgentInitializer`. The given parameter `nbOfAgentsPerNode` specifies how many `Agents` are registered and started on an `OcmNode` before a new `OcmNode` is created.

```
public TEMAgentInitializer(int nbOfAgentsPerNode) {...}
```

Important Methods

- **initializeAgent:** Registers and starts the `agent` on an `OcmNode`. `initData` is used to initialize the `agent` by calling its method `initAgent(initData)` and might be `null` if no such data is required. The TEM automatically calls the `agent`'s `init` method. If the number of agents running on the current node equals the specified maximum number of agents per node, a new node is created.

  If the `TEMAgentInitializer` is set as `IConcreteAgentInitializer` when initializing the `AgentInitializer` (see Section 4.2.1), this method is automatically called when the method `initializeAgent(...)` is called on the `AgentInitializer`.

  ```
  public void initializeAgent(Agent agent, Map<String, Serializable> initData)
      {...}
  ```

Usage This `AgentInitializer` should be used to initialize `Agents` if the system is to be executed in a **local** TEM execution environment. For an example, see Section 4.2.1. If you want to use the distributed TEM execution environment, the `TEMAgentInitializer` has to be modified since, in a distributed TEM execution environment, only a single `OcmNode` per Java VM is allowed (see Section 4.3).

4.2.4 TemLightAgentInitializer

```
public class TemLightAgentInitializer implements IConcreteAgentInitializer
```

This class is an implementation of the interface `IConcreteAgentInitializer` (see Section 4.2.2). It can be used to initialize `Agents` in the temLight simulation environment.

Important Methods

- **initializeAgent:** Initializes underlying concepts that enable message transport in the temLight simulation environment, the underlying trust functionality provided by the TEM if `agent` is a `TrustAgent`, and calls the `agent`'s `init` method. Again, `initData` is used to initialize the `agent` by calling its method `initAgent(initData)` and might be `null` if no such data is required.

  If the `TemLightAgentInitializer` is set as `IConcreteAgentInitializer` when initializing the `AgentInitializer` (see Section 4.2.1), this method is automatically called when the method `initializeAgent(...)` is called on the `AgentInitializer`.

  ```
  public void initializeAgent(Agent agent, Map<String, Serializable> initData)
      {...}
  ```

Usage This `AgentInitializer` should be used to initialize `Agent`s if the system is to be executed in the temLight simulation environment. For an example, see Section 4.2.1.

4.3 Execution Models

Depending on whether an application is run in the TEM execution environment or temLight simulation environment, there are two possible execution models: The *synchronous* execution model ensures that, in each time step, the `Agents`' `achieveGoals()` method is called one after another, so that an `Agent`'s `achieveGoals()` method is called if and only if its predecessor's `achieveGoals()` method has finished processing in this time step, i.e., if it returned `void`. This execution model is available for both TEM execution environment and temLight simulation environment. In case of the TEM execution environment, synchronous execution is realized by an additional agent that repeatedly informs the other agents to call their `achieveGoals()` method by sending an corresponding message to them (see Section 4.4). The *asynchronous* execution model allows concurrent execution of agents. This execution model is *only* available in the TEM execution environment. The `Agents`' `achieveGoals()` method is triggered directly by the TEM with a frequency as stated by `Agent`'s method `getInterval()`.

Table 3 states the availability of the execution models depending on whether the TEM or temLight simulation environment is used. The TEM execution environment can be deployed locally on a single or in a distributed manner on multiple computers. The temLight simulation environment can only be deployed on a single computer. If the TEM execution environment is deployed locally, multiple nodes can be hosted on a single machine. The concrete number depends on the available resources. If the TEM execution environment is deployed in a distributed manner, it is still possible to host multiple nodes on a single machine, but it is mandatory that each node is started in its own JVM.

	local TEM	distributed TEM	temLight
Synchronous	×	×	×
Asynchronous	×	×	
Number of Nodes	> 0 per computer	1 per JVM and computer	> 0 per computer

Table 3: Execution models: the TEM execution environment can be deployed on a single (local TEM) or multiple (distributed TEM) computers, whereas the temLight simulation environment (temLight) can only be deployed locally.

4.4 Agent Scheduling in the TEMAS

As explained in Section 4.3, there are several ways to schedule agents in the TEMAS. If your system should be deployed in the TEM execution environment in an asynchronous fashion, the scheduling concepts presented in this section are irrelevant. In such a setting, the TEM execution environment automatically schedules your agents by periodically calling the agents' `step` method (see Section 2.4 and Figure 5). Otherwise, in case of a synchronous execution model in the TEM execution environment, you have to use the `SyncTEMSchedulerAdapter`, and, in case of a synchronous execution model in the temLight simulation environment, you have to use the `SyncTemLightSchedulerAdapter`. In the following subsections, we introduce the concepts necessary for the scheduling of agents. These are also depicted in Figure 15.

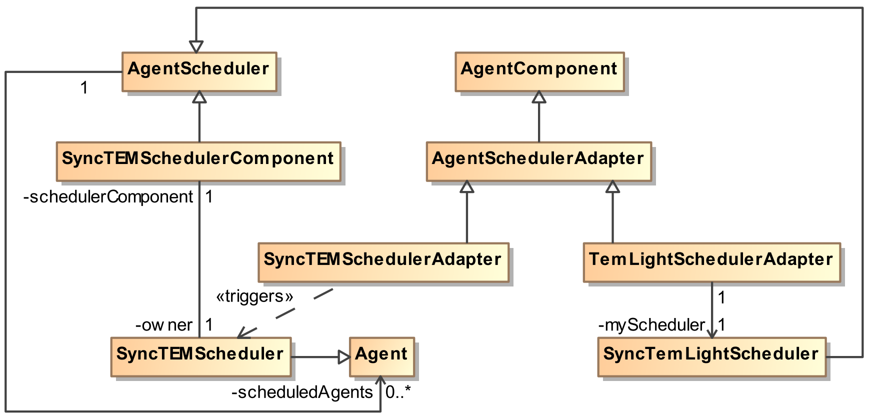

Figure 15: Concepts for agent scheduling

4.4.1 AgentScheduler

```
public abstract class AgentScheduler<T>
```

This is an abstract class that allows to schedule `Agents` in a sequential fashion. The `AgentScheduler` provides the option to schedule a list of `Agents` before another list of `Agents`. Moreover, it can be specified whether or not these two lists of `Agents` should be shuffled initially and/or periodically in order to schedule `Agents` in changing order.

Generic Type Parameters

- **T:** The type of information used to represent and schedule an `Agent`.

Constructors Creates a new instance of an `AgentScheduler`. Agents contained in the list `agents1` are scheduled before agents contained in the list `agents2`. The `boolean` parameters indicate whether the corresponding list of agents should be shuffled initially before the first call of `triggerAgents()` and/or periodically whenever `triggerAgents()` is called.

```
public AgentScheduler(boolean shuffle1Initially, boolean shuffle1Periodically,
    boolean shuffle2Initially, boolean shuffle2Periodically, List<T> agents1,
    List<T> agents2) {...}
```

Important Members

- **agents1:** A list of agents that should be scheduled before agents contained in `agents2` are scheduled.

```
protected List<T> agents1;
```

- **agents2:** A list of agents that should be scheduled after all agents contained in list `agents1` were scheduled.

```
protected List<T> agents2;
```

- **shuffle1Periodically:** Indicates whether or not to shuffle `agents1` whenever the method `triggerAgents()` is called.

```
protected boolean shuffle1Periodically;
```

- **shuffle2Periodically:** Indicates whether or not to shuffle `agents2` whenever the method `triggerAgents()` is called.

```
protected boolean shuffle2Periodically;
```

Important Methods

- **triggerAgents:** Schedules all agents contained in `agents1` and `agents2`. Agents contained in `agents1` are scheduled before agents contained in `agents2`. Lists are shuffled before the agents are scheduled as indicated by `shuffle1Periodically` and `shuffle2Periodically`. Details on how the agents are scheduled depend on the concrete implementation.

```
public void triggerAgents() {...}
```

- **addAgentAtRuntime:** Adds an agent to the `AgentScheduler` at runtime. Note that the agent is added to the end of the list `agents1`.

```
public void addAgentAtRuntime(T agentToAdd) {...}
```

- **removeAgentAtRuntime:** Removes an agent from the `AgentScheduler` at runtime. Note that the agent is removed from `agents1`. If it was not contained in `agents1`, the list is left unchanged.

```
public void removeAgentAtRuntime(T agentToRemove) {...}
```

Usage This class is extended by the `SyncTemLightScheduler` (see Section 4.4.3) and the `SyncTEMSchedulerComponent` (see Section 4.4.4) which use different ways of scheduling the agents.

4.4.2 AgentSchedulerAdapter

```
public abstract class AgentSchedulerAdapter extends AgentComponent<Agent>
```

The abstract class `AgentSchedulerAdapter` is an `AgentComponent` and thus able to send and receive messages. Each `AgentSchedulerAdapter` has a form of reference to an underlying `AgentScheduler` that actually takes over responsibility for scheduling agents. The `AgentSchedulerAdapter`'s purpose is to hide the differences of variants of the `AgentScheduler` (see Section 4.4.3 and Section 4.4.4) as best as possible. Each subclass of `AgentScheduler` should thus bring its own concrete `AgentSchedulerAdapter`. Thus, it is possible to easily test different implementations of an `AgentSchedulers` by replacing the corresponding concrete `AgentSchedulerAdapter`.

Constructors Creates a new `AgentSchedulerAdapter` with the given `owner`. The `owner` might be `null` if the concrete `AgentSchedulerAdapter` does not have to send or receive messages. When creating a new `AgentSchedulerAdapter` a form of reference to an underlying `AgentScheduler` must be set.

```
public AgentSchedulerAdapter(Agent owner) {...}
```

Important Methods

- **addAgentAtRuntime:** Informs the underlying `AgentScheduler` to add the agent `ra`. This means that the `AgentSchedulerAdapter` somehow calls the underlying `AgentScheduler`'s method `addAgentAtRuntime(ra)`.

  ```
  public abstract void addAgentAtRuntime(IAgent ra);
  ```

- **removeAgentAtRuntime:** Informs the underlying `AgentScheduler` to remove the agent `ra`, i.e., this `AgentSchedulerAdapter` somehow calls the underlying `AgentScheduler`'s method `removeAgentAtRuntime(ra)`.

  ```
  public abstract void removeAgentAtRuntime(IAgent ra);
  ```

Usage This class is extended by the `TemLightSchedulerAdapter` (see Section 4.4.3) and the `SyncTEMSchedulerAdapter` (see Section 4.4.4) which both build upon different implementations of `AgentScheduler`.

4.4.3 Scheduling in the temLight Simulation Environment

The temLight simulation environment provides synchronous scheduling of agents on the basis of the `TemLightSchedulerAdapter` that builds upon a `SyncTemLightScheduler`. The `SyncTemLightScheduler` is a concrete implementation of `AgentScheduler`. It directly calls the `achieveGoals()` method of registered `Agents` in the order defined by the `AgentScheduler`.

When the temLight simulation environment is to be deployed in combination with Repast Simphony, an additional class, called `RepastScheduler`, has to be defined in the Repast Simphony Project. The `RepastScheduler` class is needed as Repast can not schedule methods in classes that are defined outside the Repast Simphony Project. The `RepastScheduler` can be defined as follows:

```java
// Wrapper class to enable scheduling in the temLight simulation environment
// in combination with Repast Simphony.
public class RepastScheduler {
  // An instance of the TemLightSchedulerAdapter used for scheduling.
  private final TemLightSchedulerAdapter mySchedulerAdapter;

  public RepastScheduler(TemLightSchedulerAdapter mySchedulerAdapter) {
    this.mySchedulerAdapter = mySchedulerAdapter;
  }

  // Triggers the underlying TemLightSchedulerAdapter
  public void triggerAgents() {
    this.mySchedulerAdapter.triggerAgents();
  }
}
```

The idea is that Repast periodically triggers the method `triggerAgents()` of an instance of the `RepastScheduler`, which in turn calls the `triggerAgents()` method of the underlying `TemLightSchedulerAdapter`. Scheduling can then be set up in Repast in the following way:

```java
// Setting up agent scheduling in Repast Simphony
ISchedule schedule = RunEnvironment.getInstance().getCurrentSchedule();
ScheduleParameters stepParams = ScheduleParameters.createRepeating(1d, 1d,
                                          ScheduleParameters.FIRST_PRIORITY);
// Create a TemLightSchedulerAdapter
TemLightSchedulerAdapter = new TemLightSchedulerAdapter(null, true, true,
                                          false, false, agents1, agents2);
// Create a RepastScheduler
RepastScheduler repastScheduler = new RepastScheduler(mySchedulerAdapter);
// Inform Repast to schedule the repastScheduler by calling "triggerAgents"
schedule.schedule(stepParams, repastScheduler, "triggerAgents");
```

4.4.4 Scheduling in the TEM Execution Environment

The TEM execution environment provides synchronous scheduling of agents on the basis of the `SyncTEMSchedulerAdapter` that builds upon an instance of the `SyncTEMScheduler`. The `SyncTEMScheduler` is an `Agent` that incorporates a `SyncTEMSchedulerComponent` – a concrete implementation of the `AgentScheduler` – to schedule `Agents`. Synchronous scheduling is realized in the TEM execution environment as explained in the following. In synchronous scheduling mode, the TEMAS ensures that – apart from the `SyncTEMScheduler` – calling the `Agent`'s `step()` method has no effect. Whenever the `SyncTEMScheduler`'s `step()` method is triggered by the TEM execution environment's inherent asynchronous scheduling, it uses its `SyncTEMSchedulerComponent` to sequentially call the `achieveGoals()` method of registered `Agents` in the order defined by the `AgentScheduler`.

Synchronous scheduling is set up in the TEM execution environment as follows:

```
// Create a TEMSchedulerComponent which automatically creates
// its own SyncTEMScheduler
SyncTEMSchedulerComponent temSC = new SyncTEMSchedulerComponent(true, true,
                            false, false, agents1Identifier, agents2Identifier);
```

Note that the `SyncTEMSchedulerComponent` creates a `SyncTEMScheduler` on its own and that there must be exactly one `SyncTEMSchedulerComponent` in a synchronous TEM execution environment.

4.5 Bootstrapping

There are different ways to bootstrap a system that is deployed on the basis of the temLight simulation environment or the TEM execution environment. In case a TEMAS-based application is to be deployed locally, bootstrapping is rather straight forward: new nodes are created as needed on which services and agents can be started as desired. This is possible since all nodes run on a single machine. When using the TEM execution environment in a distributed environment, there are generally two ways for boot strapping:

1. **Every node is responsible for its services:** Start the service on their respective nodes. When starting an `OcmNode` all required services can be registered and started programmatically on it, using the `registerService` and `startService` methods.

2. **One node starts the services for all nodes:** When using a single node to initialize all services, an `OcmNode` provides the interface `RemoteControl` for methods to remotely register, start, stop and unregister services on other nodes. The `OcmNode` provides the method `getRemoteControl()` to obtain this interface.

In both cases, the TEMAS ensures that the agents' `achieveGoals()` method is **not** triggered until the agent receives a message with the primitive `enableStepMethod`.

RemoteControl

```
public interface RemoteControl
```

The interface to handle all remote service operations.

Important methods

- **registerServiceOnRemoteNode**: This method instantiates a service by reflection on the target node identified by the given node id and registers the service on it. The given `RemoteResult` object is notified of the result of the remote registration, either success or failure. This notification happens asynchronously, since messages have to be sent to the target node. The generated service id is returned. There exist several versions of the method, where some parameters can be omitted. The most detailed method is:

```
public <I extends Map<String, Serializable> & Serializable>
    String registerServiceOnRemoteNode(
        String fullyQualifiedClassName, String serviceType, I initialData,
        Serializable[] constructorData, String destinationNodeId,
        RemoteResult remoteResult
    );
```

The parameters in detail:

- **fullyQualifiedClassName**: The fully qualified class name of the service to instantiate.
- **serviceType**: The type of the service. This is needed to create the service id. If omitted (by calling the respective method), the `fullyQualifiedClassName` parameter is used as type. The type returned by the instantiated service must be identical to the one given here, otherwise the registration fails.
- **initialData**: The `initialData` map for the `init` method. This can be `null`. The type `I` of this parameter expects the map to be serializable.
- **constructorData**: The parameter of the constructor to call. If no constructor with the given parameters is found, the registration fails. There can only be objects as constructor parameters, i.e., no primitive types. Their respective object classes have to be used instead. If this parameter is omitted (by calling the respective method), the default constructor is used.
- **destinationNodeId**: The id of the node, where the service shall be started.
- **remoteResult**: The result object to notify, whether the registration was a success or not.

- **startServiceOnRemoteNode**: This method starts a registered service on a remote node. The service id that was created by the `registerServiceOnRemoteNode` method has to be used for this method. The result of the operation, whether the start was successful or not, is notified to the `RemoteResult` object.

```
public void startServiceOnRemoteNode(String remoteServiceId,
    String destinationNodeId, RemoteResult remoteResult);
```

- **stopServiceOnRemoteNode**: This method stops a started service on a remote node. The service id that was created by the `registerServiceOnRemoteNode` method has to be used for this method. The result of the operation, whether the stop was successful or not, is notified to the `RemoteResult` object.

```
public void stopServiceOnRemoteNode(String remoteServiceId,
    String destinationNodeId, RemoteResult remoteResult);
```

- **unregisterServiceOnRemoteNode**: This method unregisters a stopped service on a remote node. The service id to use for this method is the one created by `registerServiceOnRemoteNode`. The result of the operation, whether the unregister was successful or not, is notified to the `RemoteResult` object.

```
public void unregisterServiceOnRemoteNode(String remoteServiceId,
    String destinationNodeId, RemoteResult remoteResult);
```

RemoteResult

```
public interface RemoteResult
```

This interface is used for notification of the result of remote operations. Every application has to write its own `RemoteResult` object to react to the events.

Important methods

- **remoteActionSucceded**: When a remote operation succeeded, this method is asynchronously called. It provides the service id of the service that was remotely registered, started, stopped or unregistered as well as the node id of the node, where that operation took place.

  ```
  public void remoteActionSucceded(String serviceId, String destinationNodeId);
  ```

- **remoteActionFailed**: If a remote operation failed, this method is asynchronously called. Beside the service id and node id, it also provides the Exception that lead to the failure of the operation.

  ```
  public void remoteActionSucceded(String serviceId, String destinationNodeId,
      Exception exception);
  ```

Usage As an example, assume we have a `Service` with fully qualified class name `de.octrust.service.MyService` that we want to start on `node2`. The `Service` requires an `Integer` as constructor parameter but no `initialData` for the `init(...)` method.

```
public class MyService implements Service {

    public MyService(Integer myInt} {
        ...
    }
    ...
}
```

Note that it is important to use the `Integer` class instead of the `int` primitive type. Otherwise, the instantiation of the service will not work because Java reflection is used. We now also need a `RemoteResult` object to asynchronously receive the result of the remote registration.

```
public class MyResult implements RemoteResult {

    public void remoteActionSucceded(String serviceId) {
        System.out.println("It worked!");
    }

    public void remoteActionFailed(String serviceId, Exception exception) {
        System.out.println("Oh no, registering failed!");
```

```
        exception.printStackTrace();
    }
}
```

Under the assumption that we already have an instance of a node named `localNode`, we can now obtain the `RemoteControl` interface from the `OcmNode` and call the method `registerServiceOnRemoteNode(...)` on it.

```
RemoteControl control = localNode.getRemoteControl();
RemoteResult myResult = new MyResult();
control.registerServiceOnRemoteNode(
    "de.octrust.service.MyService",
    "de.octrust.service.MyService",
    null,
    "node2",
    new Serializable[]{2},
    myResult
);
```

`MyService` is then initialized with `myInt` = 2 and registered on `node2`. After some time, the `RemoteResult` object is notified whether or not the operation was successful.

4.6 Configuring OCµ and the TEM

The TEM can be configured by the use of three different configuration files:

- ocm.properties: allows to configure OCµ-specific properties
- tem.properties: allows to configure trust-related properties
- log4j.xml: allows to configure the logging framework used by the TEM

In the following sections, we have a closer look at these configuration files.

4.6.1 ocm.properties

The `ocm.properties` file defines all properties that are specific to OCµ. They include the following properties:

- Properties for JXTA
  - **ocm.jxta.tcp.port**: Defines the port on which JXTA listens for incoming messages.
  - **ocm.jxta.principal, ocm.jxta.password**: A user name and password internally used by JXTA, these should not be changed.
  - **ocm.transportConnector**: Defines which `TransportConnector` to use when invoking the TEM with the provided TEM.java. If using a self written main method, this property has no effect.
- Properties for the `FailureDetector`

- **ocm.failuredetector.enabled**: This property defines whether or not a TEM node should start the `FailureDetectorService` on creation. If set to `true`, the failure detector is started, if set to `false` it is not.
- **ocm.failuredetector.amount_monitoring**: Defines the minimal number of other nodes this node is monitoring.
- **ocm.failuredetector.amount_monitored**: Defines the number of other nodes this node wants to be monitored by.
- **ocm.failuredetector.heartbeat_interval**: Defines the maximal time in milliseconds which can elapse between two heartbeats until a heartbeat is considered delayed or lost.
- **ocm.failuredetector.heartbeat_samplesize**: Defines the amount of heartbeat samples to use for the failure calculation.
- **ocm.failuredetector.threshold**: Defines the threshold, when the probability of a node failure is high enough to be sure enough to warrant a broadcast that the node has failed. This is a value between 0 and 1.
- **ocm.failuredetector.waitingtime**: Defines the time in seconds between broadcasts to find suitable monitors in case the required number of monitors were not found in one period.

4.6.2 tem.properties

The `tem.properties` file includes all properties that are trust-related.

- **tem.persistence.database**: With this property the database that stores all experiences can either be file based (`file`) or an in-memory database (`memory`).
- Properties for the reputation metric, all values are between 0 and 1:
  - **tem.reputation.maxAdjustment**: The maximal change of the weight per transaction (θ in [4]), a value between 0 and 1.
  - **tem.reputation.positiveThreshold**: Threshold for positive weight adjustments (τ). When a direct experience differs between ones own experience and the one given by a neighbor by a maximum of this amount (a value between 0 and 1), then the weight gets increased and the opinion of that interaction partner is weighted higher in the next reputation request.
  - **tem.reputation.negativeThreshold**: Threshold for negative weight adjustments (τ^*). When the direct experience differs between ones own experience and the one given by a neighbor more than by the aforementioned amount but less than by this amount, the weight gets reduced, but less than defined per `tem.reputation.maxAdjustment` (This value must be greater that `tem.reputation.positiveThreshold` and between 0 and 1).
  - **tem.reputation.initial_weight**: The initial value to start for the weight, a value between 0 and 1.

- Properties for the Delayed-Ack Plugin:
  - **tem.delayed_ack.timeout**: Time in milliseconds until a message is considered as not received, when no return message with an ack arrives.
  - **tem.delayed_ack.interpreter**: The trust metric to use, either a normal mean metric (`mean`), a weighted mean with newer values weighted higher (`weighted_mean`) or a weighted mean metric with older values weighted higher (`inverted_weighted_mean`).
  - **tem.delayed_ack.experiences_amount**: The amount of experiences to use for the trust calculation (`history_length`).
- Properties for the confidence metric:
  - **tem.confidence.weight.number**: The weight for the *number confidence* in the total confidence calculation (a positive double).
  - **tem.confidence.weight.age**: The weight for the *age confidence* in the total confidence calculation (a positive double).
  - **tem.confidence.weight.variance**: The weight for the *variance confidence* in the total confidence calculation (a positive double).
  - **tem.confidence.threshold.number**: The threshold for the *number confidence*, when enough experiences are gathered (a positive integer).
  - **tem.confidence.threshold.age.recent**: A threshold in milliseconds, up until experiences are considered up to date and have an *age confidence* value of 1.
  - **tem.confidence.threshold.age.outdated**: A threshold in milliseconds, after that experiences are considered out of date and have an *age confidence value* of 0. Please note that this threshold has to be higher than the threshold `tem.confidence.threshold.age.recent`.

4.6.3 log4j.xml

In this file the logging of the TEM is defined, using log4j[7] as logging framework. Logging can be configured per package or even class by adding appropriate `<category>` tags, similar to the ones already defined. The log levels of log4j are (in ascending order of versatility, the top with least output and the bottom with the most output):

- FATAL
- ERROR
- WARN
- INFO

[7] `https://logging.apache.org/log4j/2.x/manual/configuration.html`

- DEBUG
- TRACE

4.6.4 VM Arguments

Beside the configuration files, the following VM properties can be used when starting the TEM:

- **-Docm.node.id=[self-defined id]**: By using this VM argument, the id to use for the TEM node can be defined, instead of a generated id from the `IdFactory`. This VM argument is considered by `IdFactory` implementations. For example, the `AscendingIdFactory` and `IdFactoryImpl` (default) are considering this argument.
- **-Docm.organic.manager.disabled=true**: This VM argument disables the planner of the organic manager. If enabled, the planer relocates services according to their objectives.

4.7 Setting up an Eclipse Project

Please regard the following points when setting up an Eclipse project that is based on the TEMAS:

- The project TEMReadyInfrastructure is based on the project MASConcepts.
- The Project TEMReadyInfrastructure has to be included in the build path of the project that should be based on the TEMAS
- The TEMReadyInfrastructure includes the TEM as well as the OCµ libraries and exports them so that a TEMAS-based application has access to TEM- and OCµ-related classes. Further, a `log4j` library is included.
- OCµ and TEM sources can be found on `https://swt.informatik.uni-augsburg.de:8443/artifactory/webapp/browserepo.html` under "Artifacts" and then under "libs-releases-local" or "libs-snapshots-local" for snapshot releases.

If an application is executed in the TEM execution environment and debugged in eclipse's debug mode, methods or methods that call other methods in which a message is sent and a reply awaited afterwards must not be stepped over with Eclipse's "step over" functionality but with "resume". Otherwise, a deadlock occurs.

5 Conclusion

In this technical report, we presented the Trust-Enabling Multi-Agent System (TEMAS), a multi-agent system (MAS) for open environments. The TEMAS provides basic MAS concepts such as communication and feeling for time. Moreover, the TEMAS defines concepts that allow agents to assess the trustworthiness of others by defining application-specific trust metrics and gathering experiences gained in interactions. By making informed decisions that take the trustworthiness of other agents into account, the participants of a TEMAS-based system can thus deal with uncertainties at runtime.

The TEMAS has a two-layered architecture that consists of the Trust-Enabling Middleware on the lower layer and the MASConcepts4TEM on the upper layer. While the TEM is a full-fledged middleware on the basis of a node-centric infrastructure, the MASConcepts4TEM specify additional concepts common in MAS on the one hand, and, with regard to the TEM, serves as a facade by hiding the complexity of the underlying infrastructure consisting of nodes and services and dependent interfaces to higher level applications on the other hand. In this technical report, we presented those concepts of the TEMAS which are of interest when implementing a TEMAS-based application.

In future work, we will port the TEMAS to Android-based systems, so that the infrastructure that hosts TEMAS-based applications can be equipped with mobile devices. We will further extend the TEMAS by a framework that allows to control emergent behavior of self-organizing systems. Such a framework includes a generic observer/controller architecture that allows to set up application-specific corridors of wanted or correct behavior on the basis of weak or hard constraints and define application-specific reactions that should be triggered if these corridors are left at runtime (see, e.g., [9] and [13]). Based on this framework, we will provide another framework that provides generic functionality to generate norms, observe norm compliance, and sanction behavior that contradicts existing norms at runtime.

Acknowledgment

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

Index

References

[1] G. Anders, F. Siefert, J.-P. Steghöfer, and W. Reif. Trust-Based Scenarios – Predicting Future Agent Behavior in Open Self-Organizing Systems. In *Proc. of the 7th International Workshop on Self-Organizing Systems (IWSOS 2013)*, May 2013.

[2] G. Anders, J.-P. Steghofer, F. Siefert, and W. Reif. Patterns to measure and utilize trust in multi-agent systems. In *2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pages 35 –40, oct. 2011.

[3] R. Kiefhaber, G. Anders, F. Siefert, T. Ungerer, and W. Reif. Confidence as a Means to Assess the Accuracy of Trust Values . In *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 690–697, Augsburg Univ., Augsburg, Germany, 2012. IEEE Computer Society.

[4] R. Kiefhaber, S. Hammer, B. Savs, J. Schmitt, M. Roth, F. Kluge, E. André, and T. Ungerer. The neighbor-trust metric to measure reputation in organic computing systems. In *Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2011)*, pages 41 – 46, october 2011.

[5] R. Kiefhaber, B. Satzger, J. Schmitt, M. Roth, and T. Ungerer. Trust Measurement Methods in Organic Computing Systems by Direct Observation. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 105–111, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[6] R. Kiefhaber, F. Siefert, G. Anders, T. Ungerer, and W. Reif. The Trust-Enabling Middleware: Introduction and Application. Technical Report 2011-10, Universitätsbibliothek der Universität Augsburg, 2011.

[7] D. McKnight, L. Cummings, and N. Chervany. Initial Trust Formation in New Organizational Relationships. *The Academy of Management Review*, 23(3):473–490, 1998.

[8] L. Mui, M. Mohtashemi, and A. Halberstadt. A Computational Model of Trust and Reputation. In *Proc. of the 35th Hawaii International Conference on System Sciences*, pages 188–196, 2002.

[9] F. Nafz, H. Seebach, J.-P. Steghöfer, G. Anders, and W. Reif. Constraining Self-organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach. In C. Müller-Schloer, H. Schmeck, and T. Ungerer, editors, *Organic Computing — A Paradigm Shift for Complex Systems*, Autonomic Systems, pages 79–93. Springer Basel, 2011.

[10] S. Ramchurn, D. Huynh, and N. Jennings. Trust in Multi-Agent Systems. *The Knowledge Engineering Review*, 19(01):1–25, 2005.

[11] M. Roth, J. Schmitt, R. Kiefhaber, F. Kluge, and T. Ungerer. Organic Computing Middleware for Ubiquitous Environments. In *Organic Computing – A Paradigm Shift for Complex Systems*, pages 339–351. Springer Basel, 2011.

[12] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer. A lazy monitoring approach for heartbeat-style failure detectors. *Benjamin Satzger and Andreas Pietzowski and Wolfgang Trumler and Theo Ungerer*, 6:404–409, 2009.

[13] J.-P. Steghöfer, B. Eberhardinger, F. Nafz, and W. Reif. Synthesis of Observers for Autonomic Evolutionary Systems from Requirements Models. In *Proceedings of the 6th International Workshop on Distributed Autonomous Network Management Systems*. IEEE Computer Society, 2013.

[14] 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 *Proc. of the 7th International Conference on Autonomic and Trusted Computing (ATC 2010)*. Springer, Oct. 2010.

[15] Y. Wang and J. Vassileva. Trust and Reputation Model in Peer-to-Peer Networks. In *Proc. of the 3rd International Conference on Peer-to-Peer Computing*, 2003.