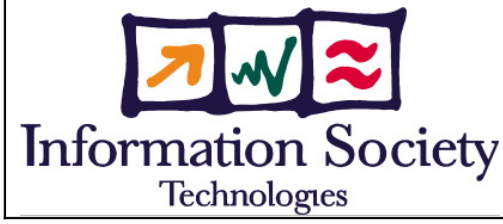
OPAALS PROJECT

Contract n° IST-034824

WP3: Autopoietic P2P Networks

Del3.11 – Implementation of Identity, Trust and Accountability Models



Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: IST-034824

Project Acronym: OPAALS

Deliverable N°: 3.11

Due date: M36

Delivery Date: M38

Short Description: This document provides a snapshot of the current status of implementation of distributed identity, trust and accountability. In the time period, and with the available resources, a higher priority was given to the development of the Identity and Trust models as these represented the most important aspects of the work given the interest and dependencies from other project members in terms of integration.

Author: Paul Malone, Mark McLaughlin, WIT

Partners contributed: WIT

Made available to: Public

Versioning

Version	Date	Name, organization
0.1	01/08/09	Initial Version Outline , Paul Malone WIT
0.2	15/0809	Identity Section complete, Mark McLaughlin, WIT
0.3	28/08/09	Trust and Accountability sections added, Paul Malone, WIT
0.4	31/08/09	Version for internal review
1	03/09/09	Final Version

Quality check

Internal Reviewers: Amir Razavi(Surrey), Paulo Siqueira (IPTI)

Dependencies:

Achievements*	This document provides a snapshot of the current status of implementation of distributed identity, trust and accountability. In the time period, and with the available resources, a higher priority was given to the development of the Identity and Trust models as these represented the most important aspects of the work given the interest and dependencies from other project members in terms of integration.
Work Packages	The work contributes to the provision of integrating accountability identity and trust in the developing platform via WP5.
Partners	IPTI (platform development), Surrey (p2p platform),
Domains	Identity, Trust, Accountability, Distributed Computing, Cryptography, Security
Targets	Other Researchers, System Implementers, SMEs, Public Administrators, Social Scientists
Publications*	Identity: Koshutanski, H., Ion, M. and Telesca, L., 2007, *A distributed identity management model for digital ecosystems*, in Proceedings of International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'07), IEEE Press and McLaughlin, M., Malone, P. and Jennings, B., A Model for Identity in Digital Ecosystems, in Proceedings of *3nd IEEE International Conference on Digital Ecosystems and Technologies*, Istanbul, Turkey, June 2009. Trust: McGibney, J. & Botvich, D., 2007. A Trust Overlay Architecture and Protocol for Enhanced Protection against Spam. In Proceedings of *The Second International Conference on Availability, Reliability and Security*. IEEE Computer Society, pp. 749-756. and McGibney, J. & Botvich, D., 2008. A trust based system for enhanced spam filtering. *Journal of Software*, 3(5), 55-64. The Trust Algorithms are published in McGibney, J. and Botvich, D., 2007, "Distributed dynamic protection of services on ad hoc and p2p networks", in *Proceedings of 7th IEEE International Workshop on IP Operations and Management (IPOM)*, San Jose, CA, USA, Lecture Notes in Computer Science (LNCS) 4786, pp 95-106, Springer, November 2007 Accountability: Malone, P. and Jennings, B., 2008, Distributed Accountability Model for Digital Ecosystems, *2nd IEEE International Conference on Digital Ecosystems*

	and Technologies, Phitsanulok, Thailand, February 2008.
PhD Students*	N/A
Outstanding features*	The use of a modeling framework to build generic identity protocols (operations), with the possibility of multiple, re-usable bindings and integration with the trust, represents a significant advance in the state of the art beyond a SAML-inspired, identity federation approach. The ongoing work on relative naming and identity, including a URI scheme and piecewise, potentially privacy preserving URI resolution, coupled with an encoding of entity-centric trust relationships into URIs, has the potential for a significant advance in a combined theory of identity, trust and naming in digital ecosystems and decentralised environments as a whole. The work performed on identity and trust by the partners in OPAALS has been published to communities outside the digital ecosystems community The work on Accountability provides an incremental change in the state of the art by providing a model and protocol to enable a service composition capable accountability framework to operate in a distributed and private manner.
Disciplinary domains of authors*	Paul Malone, Mark McLaughlin, WIT, Computer Science

Table of Contents

1 Introduction

This document provides a snapshot of the current status of implementation of distributed identity, trust and accountability. In the time period, and with the available resources, a higher priority was given to the development of the Identity and Trust models as these represented the most important aspects of the work given the interest and dependencies from other project members in terms of integration.

The distributed Identity model implementation is complete. The work provides a java implementation of the identity Operation Model and bindings for dispatchers and interceptors for JXTA and HTTP. This allows for the posibility for integration of the identity Operation Model in heterogeneous transport environments. The bulk of the work that remains in terms of Identity is a full scale integration with FlyPeer, which is in progress. An open source project IdentityFlow is in place for retrieving code and binaries to allow third parties to leverage on the work performed.

In terms of Trust model development, the basic framework is in place. This includes the implementation of TrustManagers as well as a mechanism for configuring algorithms on a per node basis. Some sample algorithms are provided. A trust referral overlay is in place which makes use of JXTA. More work needs to be done on algorithms, particularly on the integration of usage data from accountability into experience reports for trust and the development of algorithms which can interpret this usage data to meaningful trust values. An open source project TrustFlow is in place for retrieving code and binaries to allow third parties to leverage on the work performed.

Work on the Accountability model has started. The basic work of data modelling and definition of actor and protocol interfaces is finalised. Further work on the implementation of these interfaces in FlyPeer and JXTA is continuing. The project will be available on an open source project when sufficient code is available for third parties to make use of.

2 Distributed Identity Implementation

2.1 Operation Model Implementation

The Operation Model is a meta-model for Operations, which was outlined in deliverable D3.9. The concepts of Actor, Connection, Profile, Binding and Operation were introduced as components for building generic Operations that perform identity tasks such as sign-on, sign-off, claim verification, and so on. In general, Operations can support any task whereby parties (identity providers) assert identity statements (SAML assertions) concerning parties (subjects) to other parties (relying parties). Operations coordinate the message passing between the Actors involved.

Operations may execute differently based on dynamic factors that prevail during an execution. For example, the Operation designer might specify that a relying party's identity provider will accept a subject's identity assertion from an identity provider if and only if another identity provider makes exactly the same assertion. If both make the same assertion, then the subject's identity is verified and the relying party receives a token from its identity provider, otherwise the two asserting identity providers and the subject are sent a 'sign-on failed' message. Entering an incorrect password is a contingency that must be dealt with by all sign-on solutions, but clearly, in decentralised environments, supported by networks of trust, many more kinds of contingencies must be catered for. In the example above, an operation, which may be custom built for a particular scenario (perhaps even automatically generated for the given scenario), must deal with two contingencies: 1. Both asserting IdPs pass the same assertion, and 2) Both IdP's do not pass the same assertion. One can imagine additional contingent logic being built into this Operation, such as another possible continuation of the Operation if one IdP passes an assertion but the other one fails to respond completely, and so on.

In order to build this contingent logic into Operations, we allowed for a number of custom operators to be implemented, whereby at a given point in the Operation execution, an actor, in a particular state, on receiving an incoming Connection, will perform different actions and initiate different outgoing Connections based on dynamic factors.

2.1.1 Contingency and Actor State

We implemented a basic state machine model for Operations, in order to keep track of actions Actors should be performing at each point during Operation execution. Each Actor derives its own state machine from the Operation specification. Actor states change on incoming Connections. When a new state begins, a program called a StateProcessor is called for that state, which executes a series of actions. It may operate on data received from the incoming Connection, and it may prepare a data bundle to be sent to another actor as part of a follow-on outgoing Connection. The actions

performed by the StateProcessor, the data bundle and the choice of outgoing Connection (if any), depends both on a) the dynamic conditions at the time of execution and b) the conditional logic defined by the Operation (the java implementation of this functionality is summarised in section 2.3.2). This conditional logic function essentially operates on the current state and the incoming Connection.

2.1.2 Operation Model Implementation with Contingency

For the implementation, it was necessary to support contigencies by adding concepts to those outlined in the Operation model: ExecutableUnit, Section and Segment. Sections also allow us to nest portions of the Operation, allowing for sub-Profiles and sub-Sections, giving Operation designers increased flexibility. These concepts allow us to add contingent logic to Operations, and are explained below.

ExecutableUnit

ExecutableUnits are portions of Profiles that are executed in one 'visit' to a given Profile during Operation execution. Each Profile may define N ExecutableUnits. Each ExecutableUnit executes a series of Sections (below).

Profiles, which are adapted from SAML profiles, define portions of overall protocol flow. Consider the SAML Single Sign-On (SSO) Profile given in Figure 2.1. The profile detemines some interactions between the three Actors, then hands off to some protocol flow that identifies the principal, which is undefined within the profile, and then returns to the same profile at a later stage where the final interactions are defined that complete the SSO. In our Operation Model, this entire protocol flow would be described by an Operation. The Operation would contain two profiles: the SAML SSO Profile and 'Identification' Profile. At some point during the execution, control is handed from the SSO Profile to the Identification Profile and then back again. In order to accomplish this kind of hand off we define ExecutableUnits. Each time control is handed to a given Profile instance, a counter is incremented, and the next ExecutableUnit is executed. Once the execution of that ExecutableUnit completes, control passes to the next Profile, or more specifically, the next ExecutableUnit of the next Profile.

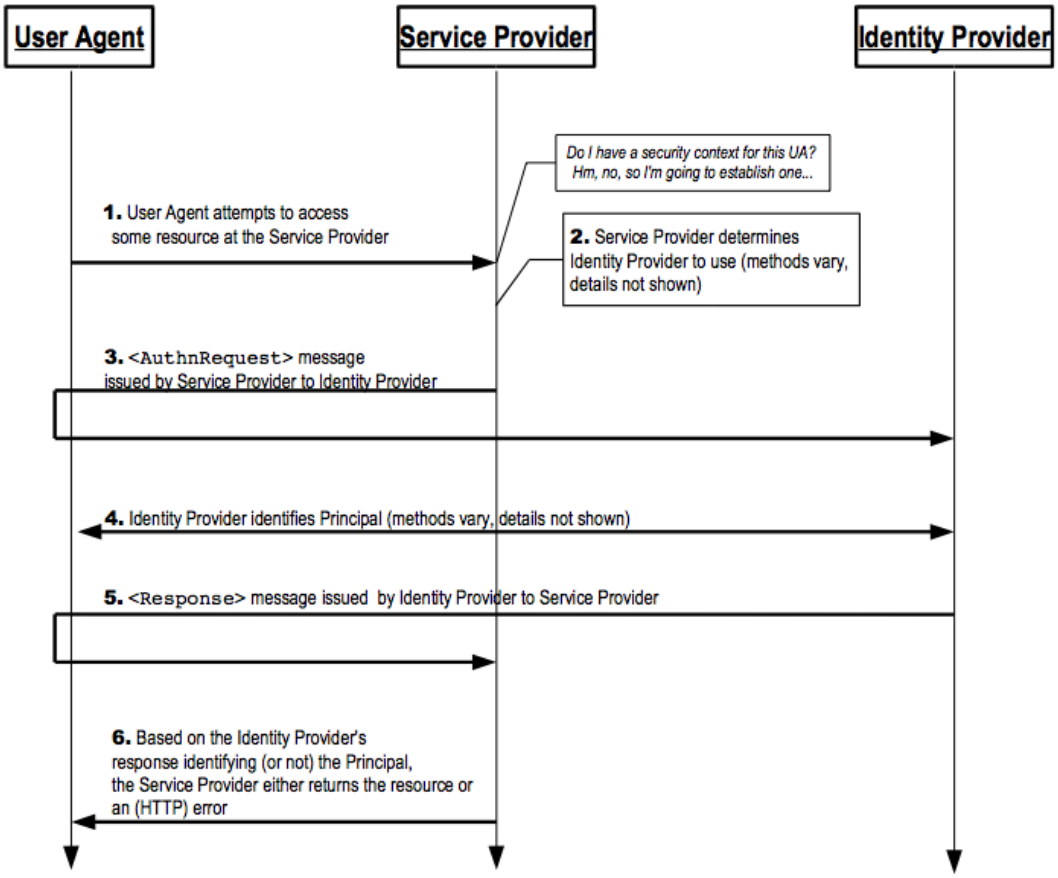


Figure 2.1: SAML Single Sign-On Profile

Section

Sections are used to implement contingent logic and nesting in Operations. Sections are the units of 'execution' that are chosen between by operators representing this logic. Sections can be decided between (OR), executed 'if and only' if some condition is true (→), executed in series (SER) or in parallel (PAR). Sections represent either a) a Profile (indirectly a particular ExecutableUnit of that Profile), b) a Segment (see below) or c) a compound Section. Compound Sections contain a List of Sections and may be one of two types: SER or PAR. SER Sections contain Sections that are executed in series, which PAR Sections contain Sections that are executed in parallel.

Segment

Segments are simply a list of Connections that are executed in series. No contingent logic is possible within a Segment.

From these three additional elements, we can make the following observations:

- A strong resemblance of Profiles to SAML Profiles has been preserved to enhance interoperability with SAML specifications and to provide a compatible representation.

- Operations are extremely configurable and should (pending a formal analysis) be capable of defining arbitrary protocol flows.
- Since Profiles (and other Sections) may be nested within Sections, which are themselves nested within the ExecutableUnits of Profiles, we can allow significant nesting of portions of the Operation, giving us additional flexibility in the Operation representation and allowing portions of the Operation to be implemented by other parties (domain experts). For example, in Figure 2.1, from the SAML specifications, the authors are only interested in defining the SAML Profile and not that portion of the protocol flow concerned with 'Identification'. When both portions can be separated into two (pluggable) Profiles, both Profiles can be designed separately by parties with the appropriate interest and expertise.

The components of the implemented Operation model are given in Figure 2.2. An illustrated Operation assembled from these components is given in Figure 2.3. See Section 2.3.1 for java implementation detail.

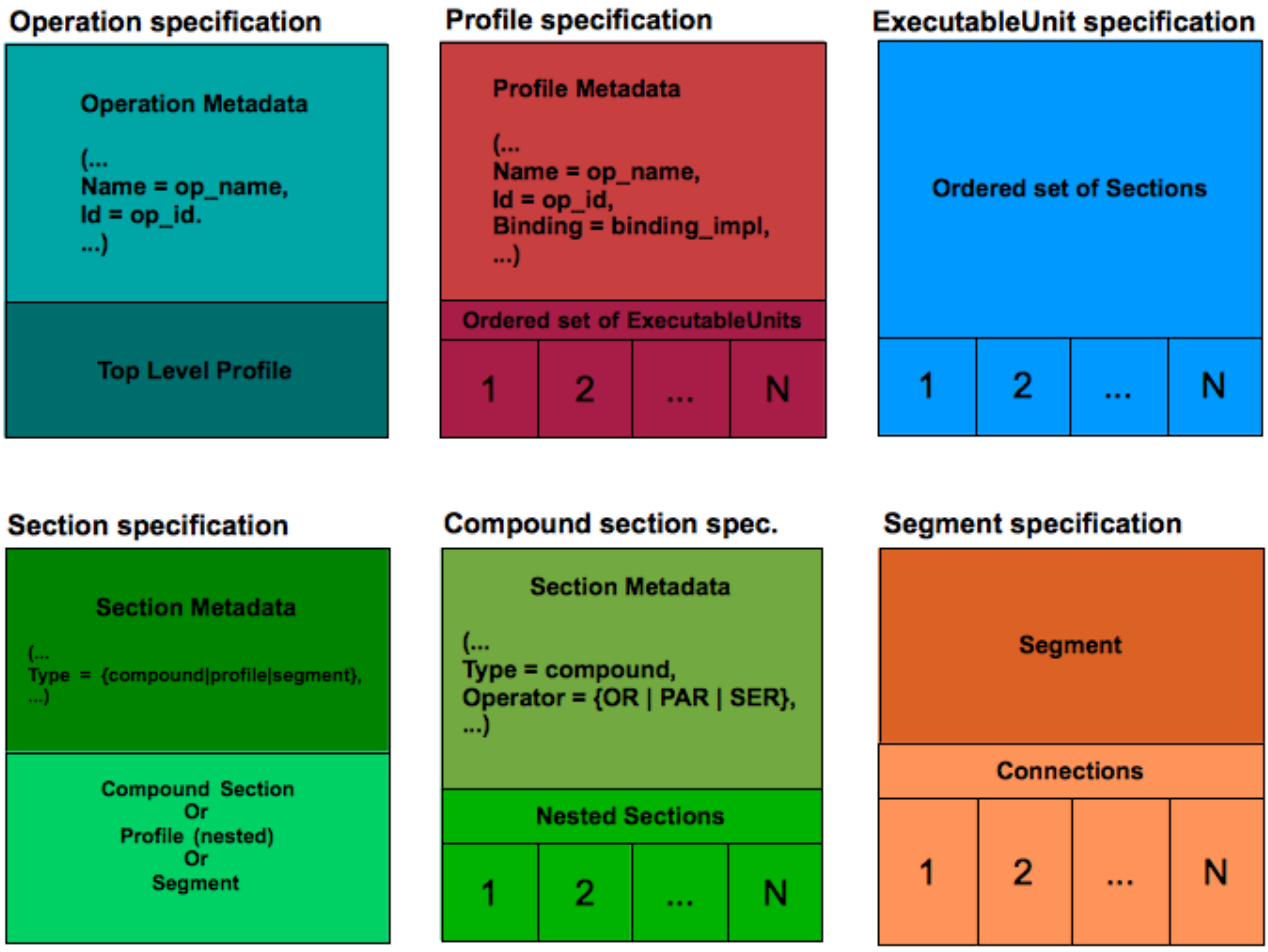


Figure 2.2: Implemented Operation Model Components

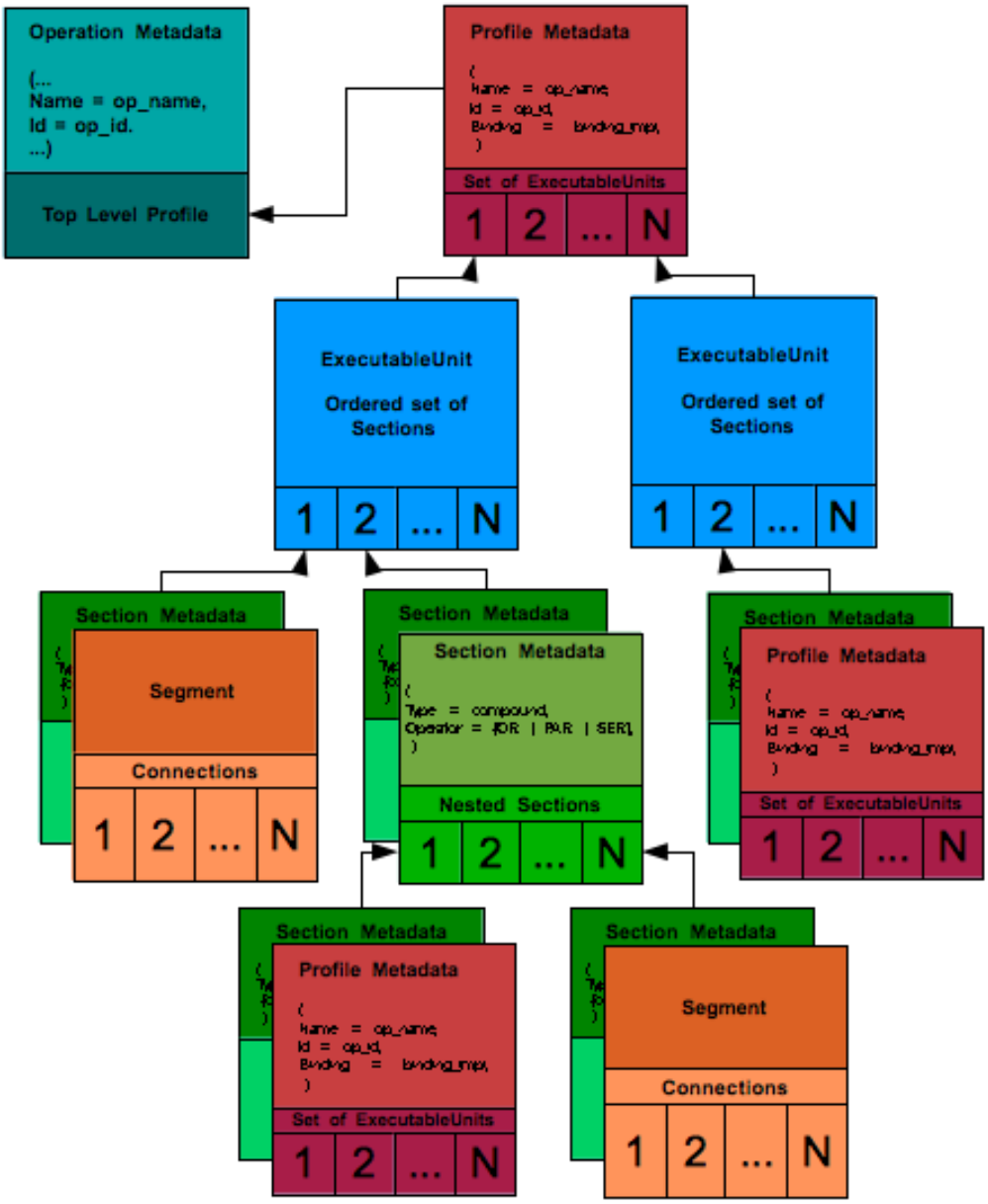


Figure 2.3: Illustrated Operation

2.2 Support for Heterogeneous Environments

The Operation Model implementation was built to support a digital ecosystem (DE) comprising heterogeneous technical infrastructures. The use of Bindings, which are analogous to SAML bindings, allow us to construct Profiles that may be executed in a particular transport environment. For example, a HTTP Redirect binding might use HTTP GET requests to pass all messages; a JXTA binding might use JXTASockets and JXTAPipes to pass all messages.

Each Profile must have an associated Binding, but since an Operation may be composed of a number of Profiles (and 'sub-Profiles'), portions of Operations can execute using different Bindings. This allows an Operation to cross infrastructural divides within a DE. Figure 2.4 illustrates a series of Operation Connections crossing infrastructural boundaries. Connections executing in each environment use a seperate Profile and Binding (a Profile may consist of a single ExecutionUnit, which may contain a single Section, containing a single Connection, such that each Connection can potentially use a different binding).

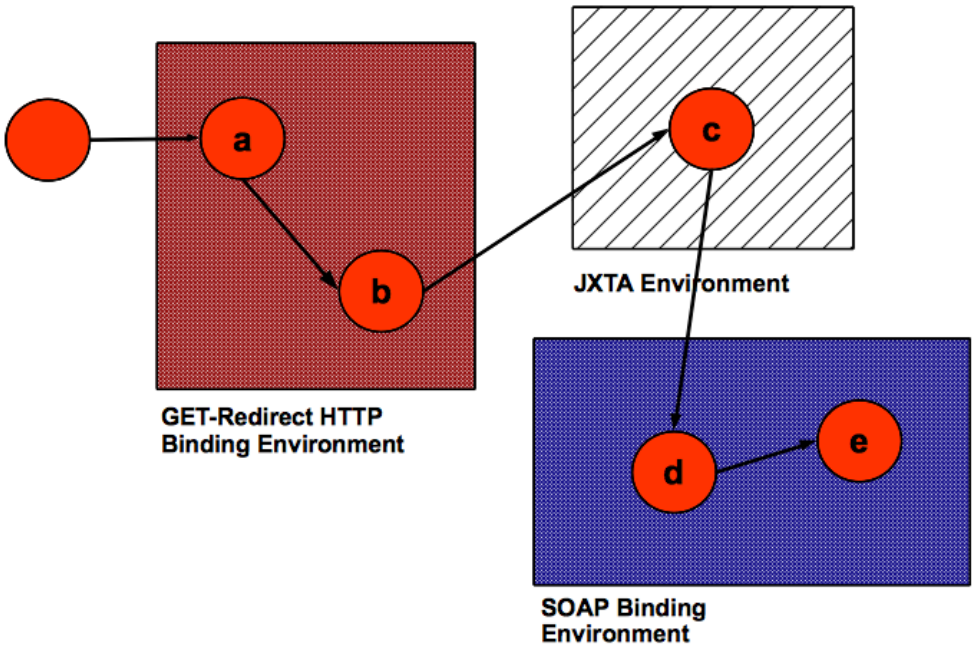


Figure 2.4: A series of Operation Connections in a heterogeneous environment

2.2.1 Dispatchers and Interceptors

Ultimately, Bindings are supported by two constructs in the implementation: Dispatchers and Interceptors. Dispatchers allow Connections to make a uni-directional transport connection and transfer data from one node to another. The type of Dispatcher used depends on the Binding. The Dispatcher implementation encapsulates the logic for making the transport connection in the given environment. For example, a GetHTTPDisptacher would open a HTTP connection to a URL on a given port of the destination node and transmit data using a 'GET' request, ie. inserting the data as attributes in the URL. A SOAP Dispatcher, using a HTTP SOAP binding, would similarly open a HTTP connection on a port to a web service container on a node, but the data would be sent in the form of an XML document. A JXTA Dispatcher would open a JXTASocket to another peer with a

given PeerID and transfer data using some arbitrary mechanism. Similarly, for each type of Binding, and hence Dispatcher, a corresponding Interceptor is used to intercept incoming Connections on the destination node. Once the relevant service/application/web container or server has received an incoming request that is identified as an IdentityFlow Operation request, the request is passed to the relevant Interceptor. A web server might pass requests to a HTTPInterceptor; a servlet container to a ServletInterceptor; a JXTA node to a JXTAInterceptor; and so on. See Section 2.3.3 for java implementation detail.

2.3 Java Implementation

In this section we give selected java code snippets to illustratrate key implementation concepts. Instructions on how to obtain and build the code can be found in Section 2.4.

2.3.1 Operation Model

An excerpt from the HandledOperation class is given in code snippet 1, which is the class that all current Operation implementations override. We see that the HandledOperation overrides NestedOperation, which allows for the nesting of Profiles (ie. the use of NestedProfiles), and specifies key attributes such as the topLevelProfile (and topLevelBinding), which is the starting point for NestedOperation execution, as well as metadata such as the operationId and the Actors involved in the Operation.

Code snippet 1:

```
public class HandledOperation<ReqParamKey,ReqParamValue> extends NestedOperation {

        private final Hashtable
                <Actor,ParameterConnectionMultiplexer<ReqParamKey,ReqParamValue>>
                        defaultMuxes;

        private final Hashtable<Actor,Hashtable<Condition,State>>
                initialConditionStates;

        public HandledOperation(
                        NestedProfile topLevelProfile,
                        Binding topLevelBinding,
                        String getOperationId,
                        Actor initiatingActor,
                        Actor[] actors,
                        Hashtable<Actor,ParameterConnectionMultiplexer
                                                        <ReqParamKey,ReqParamValue>>
defaultMuxes,
                        Hashtable<Actor,Hashtable<Condition,State>> initialConditionStates) {

                super( topLevelProfile,
                                topLevelBinding,
                                getOperationId,
                                initiatingActor,
                                actors);

                this.defaultMuxes = defaultMuxes;
```

```
                this.initialConditionStates = initialConditionStates;
        }

....
}
```

An excerpt from the HandledProfile class is given in code snippet 2, which is the class that all current Profile implementations override. ExecutableUnit, Section and Segment classes are inner classes of NestedProfile, as shown.

Code snippet 2:

```
public class NestedProfile implements Profile {

        /* Profile Id */
        private final String profileId;

        ....

        private final ExecutableProgram executableProgram =
                new ExecutableProgram();

        private int executableProgramPosition = -1;

        public NestedProfile(String profileId, Baton baton) {
                this.profileId = profileId;
                this.baton = baton;
        }

....

        public static class ExecutableUnit extends ArrayList<Section> {
                private static final long serialVersionUID = 1;

                public void buildExecutableUnitFromSequentialSections(
                                List<Section> sections) {

                        for(Section section: sections) {
                                add(section);
                        }
                }
        }

....

        public static class Section {
                public static final String SEGMENT_TYPE = "SEGMENT_TYPE";
                public static final String PROFILE_TYPE = "PROFILE_TYPE";
                public static final String COMPOUND_TYPE = "COMPOUND_TYPE";

                private final String type;
                private final boolean required;

                private Section(String type, boolean required) {
                        this.type = type;
                        this.required = required;
                }
....

        public static class CompoundSection extends Section {
```

```
                public static final int OR_TYPE = 0;
                public static final int PAR_TYPE = 1;
                public static final int SER_TYPE = 2;

                private final List<Section> sections;
                private final int compoundType;

                private CompoundSection(List<Section> sections,
                                                        int compoundType,
                                                        boolean required) {
                        ....
                }
                ....
        }

....

}
```

An excerpt from the ProperActor class is given in code snippet 3, which is the default implementation of the Actor interface.

Code snippet 3:

```
public class ProperActor implements Actor {

        private final String actorId;
        private final String instanceId;

        /**
         * Instanciates a ProperActor.
         * @param actorId
         * @param instanceId can be null if there is only one Actor instance.
         */
        public ProperActor(String actorId, String instanceId) {
                this.actorId = actorId;
                this.instanceId = instanceId;
        }
....
}
```

An excerpt from the AbstractLiveConnection class is given in code snippet 4. All current Connection implementations extends AbstractLiveConnection, which uses Dispatchers to implement transport level functionality. Current Connection implementations also add logic to map Connections to State Conditions (see Section 2.3.2).

Code snippet 4:

```
public abstract class AbstractLiveConnection<DispatcherType extends Dispatcher>
                                        extends AbstractConnection
                                        implements LiveConnection {

        protected AbstractLiveConnection(    String connectionId,
                                                String profileId,
                                                String bindingId,
                                                Actor sourceActor,
                                                Actor destinationActor) {
```

```
            super(connectionId, profileId, bindingId, sourceActor, destinationActor);
        }

        /**
         * Sets Dispatcher for this LiveConnection.
         * Dispatchers encapsulate the logic to dispatch a connection.
         * @param dispatcher
         */
        public abstract void setDispatcher(DispatcherType dispatcher);

        /**
         * Returns the Dispatcher for this LiveConnection.
         * Dispatchers encapsulate the logic to dispatch a connection.
         * @return
         */
        public abstract DispatcherType getDispatcher();

}
```

Figure 2.5 illustrates an Operation execution using the implemented components described.

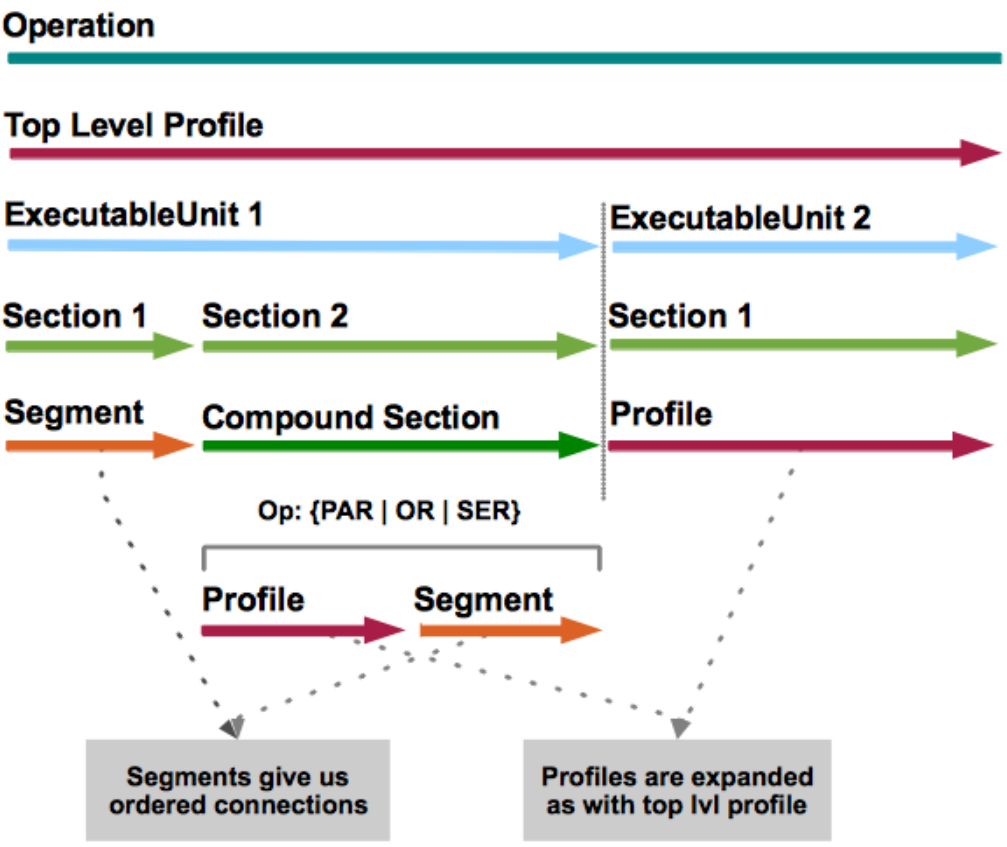


Figure 2.5: The Execution of an Operation

2.3.2 Operations, StateMachines and Contingency

We said in Section 2.1.1 that the conditional logic function that decides what each node does

following an incoming Connection essentially operates on both the current state and the incoming Connection. We have already modelled Connections in the Operation Model, however, we need a StateMachine model for Actors to keep track of their State. We have two important requirements of this StateMachine model:

1. Connections must somehow trigger State changes and give rise to some Condition that gives rise to State Transitions.
2. The StateMachine definitions used by each Actor must be derivable from the Operation definition.

We solve 1. with the following convention:

a) State changes are triggered on incoming Connections.

b) Each Connection implementation (StateTriggerConnection) is associated 1:1 with a Condition from the StateMachine model.

We solve 2. with a class called ConnectionMatrix that recurses through the NestedOperation and builds a tree of all of the possible Connections that could be made, under all circumstances. for the Operation. SER Connections ascend from node to node from the root outwards. OR and PAR Connections diverge at a given node, indicating independent lines of Operation execution. From this tree, the ConnectionMatrix can then derive for each Actor with a tree of all possible Connections that could potentially connect to that Actor. From a) and b), we have already chosen that it is incoming Connections we are interested in and that these Connections are associated with a Condition. The Actor State that is triggered by a given Connection is then produced by a function of the current State of the Actor and the associated Condition. Once the Actor assumes a given state, a program called a StateProcessor is run.

To clarify this further, we will first outline briefly the StateMachine Model, then give an illustrative example, and finally give selective code snippets from important classes.

StateMachine Model

- State: A finite state machine state.
- Transition: A finite state machine transition.
- Condition: A prevailing condition that leads to a transition.
- StateMachine: A simple finite state machine that changes state from one state to another depending on the prevailing Condition.
- StateProcessor: A program that is run at the point where an Actor assumes a given State.

Illustrative Example

Figure 2.6 is an illustration of a sample Operation. This Operation does not contain any contigencies and therefore all Connections must be executed in the given order. We give this simplified example for ease of presentation. Figure 2.7 is an illustration of the StateMachine that the ConnectionMatrix would supply to Actor B arising from this Operation. Similar StateMachines would be supplied to Actors A and C. The middle chain of execution, given by the black arrows, reflects 4 states for B: its initial 'ready' State, and the series of States arrising from the Conditions mapped to each of the 3 incoming Connections to Actor B outlined in the Operation in Figure 2.6. The other two chains of execution, represented by purple arrows, indicate other possible States arising from contingent logic that might arise from a more complex, or expanded Operation from that given in Figure 2.6, wherein there are alternative paths that the Operation execution may follow.

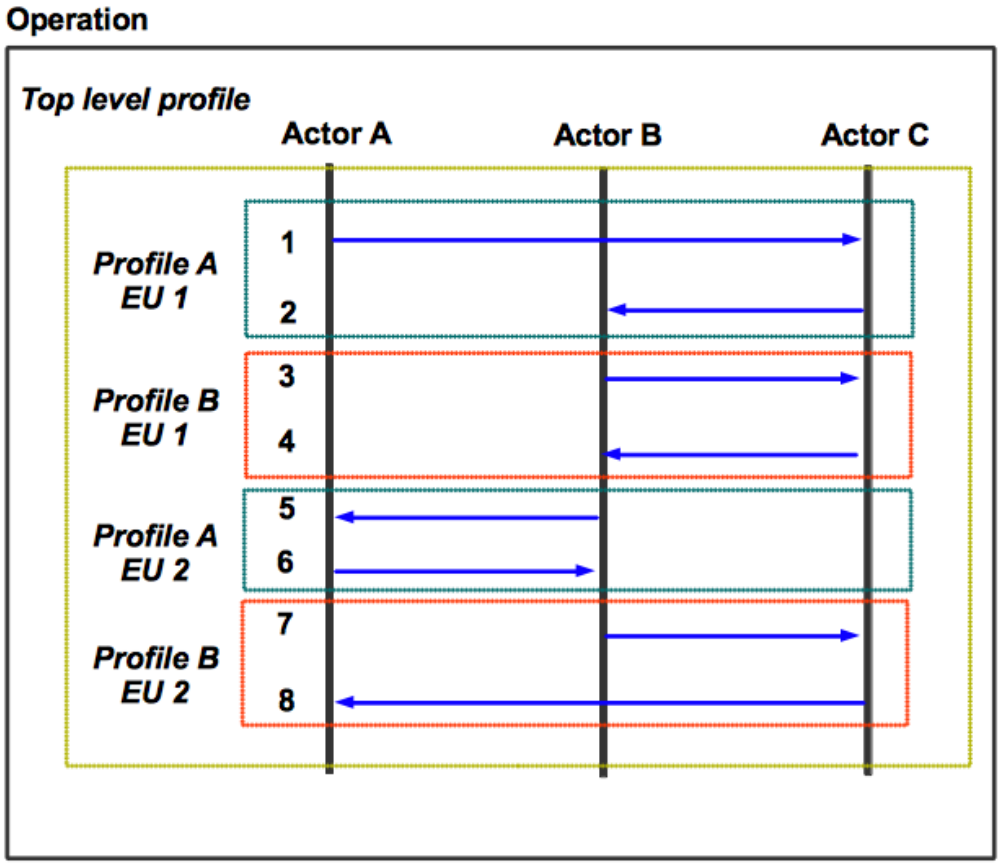


Figure 2.6: Example of an Operation Protocol Flow (with no contingency)

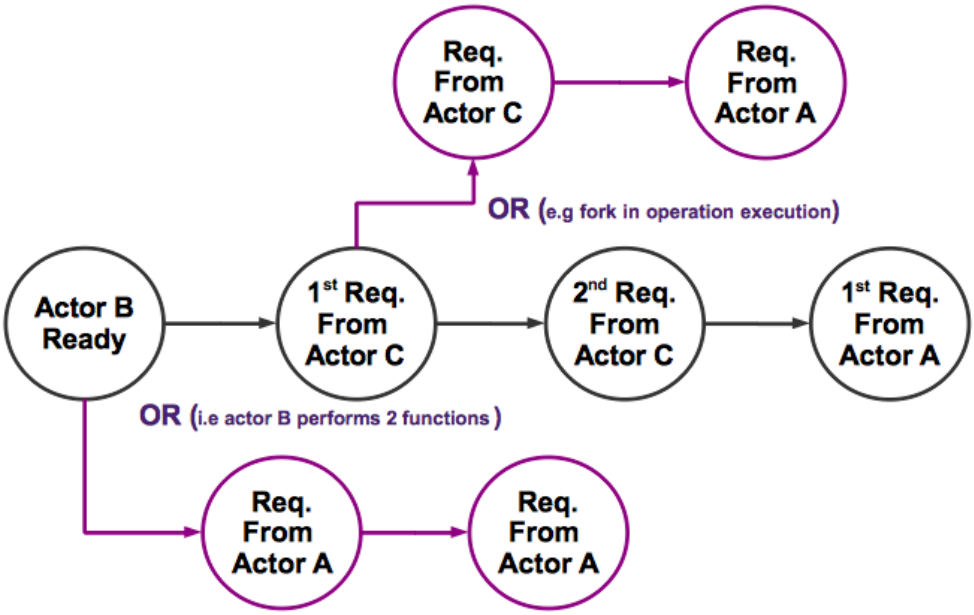


Figure 2.7: The StateMachine of Actor B derived from Operation in Figure 2.6

Code Snippets

Code snippet 1:

State and Condition implementations are merely single String member classes. Portions of the Transition and StateMachine implementations are given below.

```
public class TransitionImpl implements Transition {
	private final State current;
	private final State next;
	private final Condition change;

	public TransitionImpl(State current, State next, Condition change) {
		this.current = current;
		this.next = next;
		this.change = change;
	}

....
}

public class IncomingTriggeredStateMachine extends StateMachine {

	public IncomingTriggeredStateMachine(State startState, State successState)	{
		super(startState, successState);
	}

	private final Map<State,StateTriggerConnection<?,?>> stateConnectionMap =
			new Hashtable<State,StateTriggerConnection<?,?>>();

....
```

```
        /**
         * Adds a Transition that is triggered on an incoming Connection.
         */
        public void addTriggeredTransition(
                                            Transition transition,
                                            StateTriggerConnection<?,?> connection) {

                addTransition(transition);
                mapStateConnections(transition.getNextState(), connection);
        }

....

}
```

Code snippet 2:

Excerpts from the StateTriggerConnection class, which effectively links the Operation and StateMachine models, are given below.

```
public interface StateTriggerConnection<InputKey,InputValue>
                                                                extends LiveConnection {

        /**
         * Returns the Incoming ParameterConnectionMultiplexer.
         * The previous incoming Connection always determines the logic
         * governing the current State. When a new incoming Connection is made,
         * this Connection is determined from the request by
         * RequestConnectionMultiplexer for the current State, which is given
         * by the previous incoming Connection.
         * @return
         */
        public ParameterConnectionMultiplexer<InputKey,InputValue>
                                                            getIncomingMultiplexer();

....

        /**
         * Returns the Condition corresponding to this Connection.
         * That is the Condition that is represented by the decision to followthis
         * Connection.
         * @return
         */
        public Condition getCondition();

....

}
```

Code snippet 3:

Excerpts from the ConnectionMatrix class, which builds a tree from all possible NestedOperation contigencies and builds Actor StateMachines are given below. The ConnectionMatrix is built up by traversing the NestedOperation and attaching nodes on to a ConnectionMatrix tree using the attach() methods. This tree is then mined by recursive methods such as getNodeConnections(), which returns all node Connections for a given Actor.

```
public class ConnectionMatrix {

        ....

        /**
         * The root node.
         */
        private final Node rootNode = new Node();

        ....

        public ConnectionMatrix(Actor initiatingActor) {
                this.initiatingActor = initiatingActor;
        }

        ....

        public List<Node> attachToRoot(      Operator operator,
                                                List<Connection> connections) {

                if(!checkArgs(operator, connections)) return null;

                return attach(rootNode, operator, connections);
        }

        public List<Node> attach(     Node node,
                                        Operator operator,
                                        List<Connection> connections) {

                if(!checkArgs(operator, connections)) return null;

                node.setFollowingOperator(operator);

                List<Node> nodes = new java.util.ArrayList<Node>();

                for(Connection connection: connections) {
                        nodes.add(node.attachConnection(connection));
                }

                return nodes;
        }

        public boolean hasBranches(Node node) {
                return node.hasBranches();
        }

        public Operator getOperator(Node node) {
                return node.getOperator();
        }

        ....

        /**
         * Recursive method that spans the entire ConnectionMatrix (tree), adding
         * Connections to a List.
         * @param node is the root node.
         * @param cons is the List that Connections are added to.
         * @param actor specifies the Actor that the Connections pertain to.
         * If actor is null all Connections are included.
         */
        private void getNodeConnections(     Node node,
                                                List<Connection> cons,
                                                Actor actor) {
```

```
                Iterator<Connection> following = node.getFollowingConnections();

                if(following == null) return;

                while(following.hasNext()) {
                        Connection thisCon = following.next();

                        if(actor == null || thisCon.getDestinationActor().equals(
                                                                                actor)) {
                                cons.add(thisCon);
                        }

                        getNodeConnections(
                                        node.getTerminatingNode(thisCon), cons, actor);
                }
        }

....

}
```

2.3.3 Support for Heterogeneous Environments using Dispatchers and Interceptors

Dispatchers and Interceptors were introduced in Section 2.2.1. We see that AbstractLiveConnection, partially given in code snippet 4 in Section 2.2.1 (along with StateTriggerConnection, partially given in code snippet 2 in Section 2.3.2), which must be overriden by all (current) Connection implementations, requires a Dispatcher type to be chosen for the class. Two examples of Dispatcher code are given in code snippet 1. Interceptors are responsible for determining which Operation and incoming Connection pertains to, then the Connection is passed to the appropriate OperationHandler. The OperationHandler handles each running instance of that Operation, and has access to the Actor state for that Operation instance. An example of an Interceptor and an OperationHandler are given in code snippet 2.

Code snippet 1:

```
public class GETRedirectDispatcher extends RedirectDispatcher {

        public GETRedirectDispatcher() {
        }

        /* @see org.tssg.opaals.profile.impl.servlet.RedirectDispatcher#dispatch()
         */
        @Override
        public void dispatch() {
                /* url and HttpServletResponse must be set for dispatch. */
                try {
                        if(url == null || response == null) {
                                throw new GETRedirectDispatcherException("URL and " +
                                                "HttpServletResponse must be set for " +
                                                "GETRedirectDispatcher.");
                        }
                }
                catch(GETRedirectDispatcherException grde) {
                        grde.printStackTrace();
                }
```

```
                StringBuilder urlString = new StringBuilder(url);

                if(attributes != null && attributes.size() > 0) {
                        urlString.append("?");

                        Set<String> attrNames = attributes.keySet();
                        for(String attrName: attrNames) {
                                urlString.append(attrName);
                                urlString.append("=");
                                urlString.append(attributes.get(attrName));
                                urlString.append("&");
                        }

                        urlString.deleteCharAt(urlString.length()-1);
                }

                try {
                        response.sendRedirect(urlString.toString());
                }
                catch(java.io.IOException iox) {
                        iox.printStackTrace();
                }
        }

....

}

public class JXTADispatcher implements Dispatcher {

        private PeerAdvertisement destinationPeer = null;
        private PeerGroup peerGroup = null;
        private CommsChannel channel = null;
        private AbstractRequestParameterMap<String,String> attributes = null;

        /* (non-Javadoc)
         * @see org.tssg.opaals.profile.dispatch.Dispatcher#dispatch()
         */
        public void dispatch() {
                synchronized(this) {
                        try {
                                /* Test if dispatch is possible: minimum settings must be set. */
                                ....

                                /* Test if attributes is non-null. (Empty set is okay for now.) */
                                ....

                                /* Build PipeAdvertisement and form SocketAddress */
                                PipeAdvertisement pipeAdvertisement =
                                        PipeAdvertisementCreator.newPipeAdvertisement(peerGroup,
channel);

                                JxtaSocketAddress socketAddress = new JxtaSocketAddress(
                                                peerGroup, pipeAdvertisement, destinationPeer);

                                /* Build XML Document with attributes as Elements */
                                Set<String> attributeSet = attributes.keySet();
                                Document attributesDoc = new Document();

                                for(String attrName: attributeSet) {
                                        Element element = new Element(attrName);
                                        element.setText(attributes.get(attrName));
```

```
                                attributesDoc.addContent(element);
                        }

                        /* Grab JXTA socket and send XML Document to OutputStream */
                        JxtaSocket socket = new JxtaSocket(socketAddress);

                        OutputStream outputStream = socket.getOutputStream();
                        XMLOutputter outputter = new XMLOutputter(Format.getPrettyFormat());
                        outputter.output(attributesDoc, outputStream);

                        /* Flush and close streams (important for JxtaSocket/pipes) */
                        outputStream.flush();
                        outputStream.close();
                        socket.close();

                } catch (IOException iox) {
                        iox.printStackTrace();
                } catch(JXTADispatchException jda) {
                        jda.printStackTrace();
                }
        }
    }

....

}
```

Code snippet 2:

```
public class ServletRequestInterceptor
                extends AbstractRequestInterceptor<HttpServletRequest> {

....

public void intercept(HttpServletRequest request, JspWriter out) {

        String[] operationIds = findMatchingOperationIds(request);

        for(ServletOperationHandler operationHandler: operationHandlers) {
                if(operationHandler.getOperation()
                                .getOperationId().equals(operationIds[0])) {

                        ServletRequestParameterMap parameterMap =
                                new ServletRequestParameterMap(request);

                        /* If out is not null, then pass it on to handler. */
                        if(out == null) {
                                operationHandler.handle(operationIds[1],
                                                        actor,
                                                        parameterMap);
                        } else {
                                operationHandler.handle(operationIds[1],
                                                        actor,
                                                        parameterMap,
                                                        out);
                        }

                }
        }
    }

....
```

```
}


public abstract class StateTriggerOperationHandler<ReqParamKey,ReqParamValue>
        extends
        OperationHandler<IncomingTriggeredStateMachine,ReqParamKey,ReqParamValue> {

....

public void handle(
        String operationInstanceId,
        Actor thisActor,
        AbstractRequestParameterMap<ReqParamKey,ReqParamValue> reqParameters,
        StringBuffer output) {

....

                /* Retrieve OperationContext. */
                OperationContext context = getOperationContext(operationInstanceId);

                /* Determine current State. */
                IncomingTriggeredStateMachine machine =
                        (IncomingTriggeredStateMachine)getStateMachine(
                                        operationInstanceId);
                State currentState = machine.getCurrentState();

                ParameterConnectionMultiplexer<ReqParamKey,ReqParamValue>
                conMux;

....

                /* Change to next State determined by Condition. */
                machine.changeNextState(condition);

                /* Get new State. */
                State newState = machine.getCurrentState();

                /* Get StateProcessor for new State. */
                StateProcessor<ReqParamKey,ReqParamValue> processor =
                        operation.getStateProcessorMap(thisActor).get(newState);

                /* Process StateProcessor. */
                AbstractRequestParameterMap<ReqParamKey,ReqParamValue>
                        newReqParameters = null;
                if(output == null) {
                        newReqParameters = processor.process(reqParameters);
                }

....

                /* Launch outgoing Connection if appropriate. */
                if(newReqParameters != null) {
                        dispatch(newConnection, context, newReqParameters);
                }
....
        }
....

}
```

2.4 IdentityFlow Project Summary

IdentityFlow is an open source project intended to support the creation and deployment of arbitrary Operations, including those used to provide identity in the OPAALS digital ecosystems environment.

There is a blog containing news and updates at,

http://identityflow.sourceforge.net/

IdentityFlow is hosted on Sourceforge at,

https://sourceforge.net/projects/identityflow/

Instructions on how to obtain the code via CVS can be found here,

https://sourceforge.net/projects/identityflow/develop

IdentityFlow consists of a number of modules, while can be built, packaged and installed using maven2. The modules should be built in the following order to avoid dependency issues,

1. Identity Model SAML
2. Operation Builder
3. Operation Request Handler
4. JXTA Module
5. Single Sign-On Operation
6. Actors using SSO Example

The modules can be built from a command line by changing to the root directory of each in turn and running 'mvn clean install'. Maven2 and Java 1.5 must be installed.

3 Distributed Trust Implementation

3.1 Introduction

The OPAALS Trust Model for digital ecosystems is described in Deliverable D3.9 and is shown below in Figure 3.1 below. The approach is to use a trust overlay network for providing a community based approach to trustworthiness based on the reputation of entities. The Entity can represent a node, service, resource, a service provider or a service consumer. Each entity has a Trust Manager associated with it. The entities gain experience from interacting with other entities and publish reports of these experiences to the Trust Manager. Using a pre-defined context dependent algorithm the Trust Manager updates the entities' local trust and based on a policy of sharing trust information provides trust updates to other Trust Managers in the overlay network.

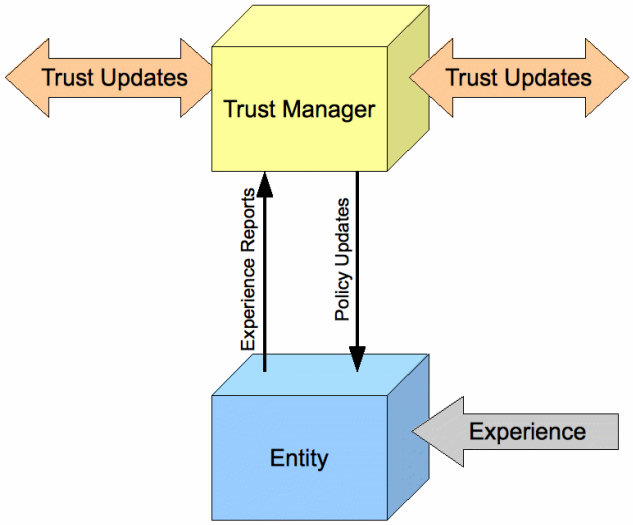


Figure 3.1: OPAALS Trust Manager

A UML Class Diagram of this model is provided in Figure 3.2. Entities are responsible for creating trust algorithms. Each Entity has an associated TrustManager. A TrustAlgorithm is associated on a per entity basis with an trustor and a context. Entities create these algorithms and publish them to the TrustManager. The TrustManager maintains a set of these algorithms and when it receives an

ExperienceReport, it uses the appropriate TrustAlgorithm by performing a lookup on trustor-context tuple.

The TrustManager also maintains a set of TrustValue objects which it updates after performing the algorithm on the ExperienceReport. The Entity can request PolicyUpdates from the TrustManager. These policy updates are derived from current TrustValues and are used by the entity in making choices about future interactions with other entities. The TrustValue class includes a 'source' element indicating whether the value was derived from direct experience or based on referrals. The class also has a timestamp attribute, as recently gathered trust might be of more value than older values. Also there is a confidence attribute, used to denote the confidence in this Trust Value (e.g. this value can be increased each time the Trust Value is updated, i.e. 100 updates to the Trust Value implies more confidence in the Trust Value than 10) .

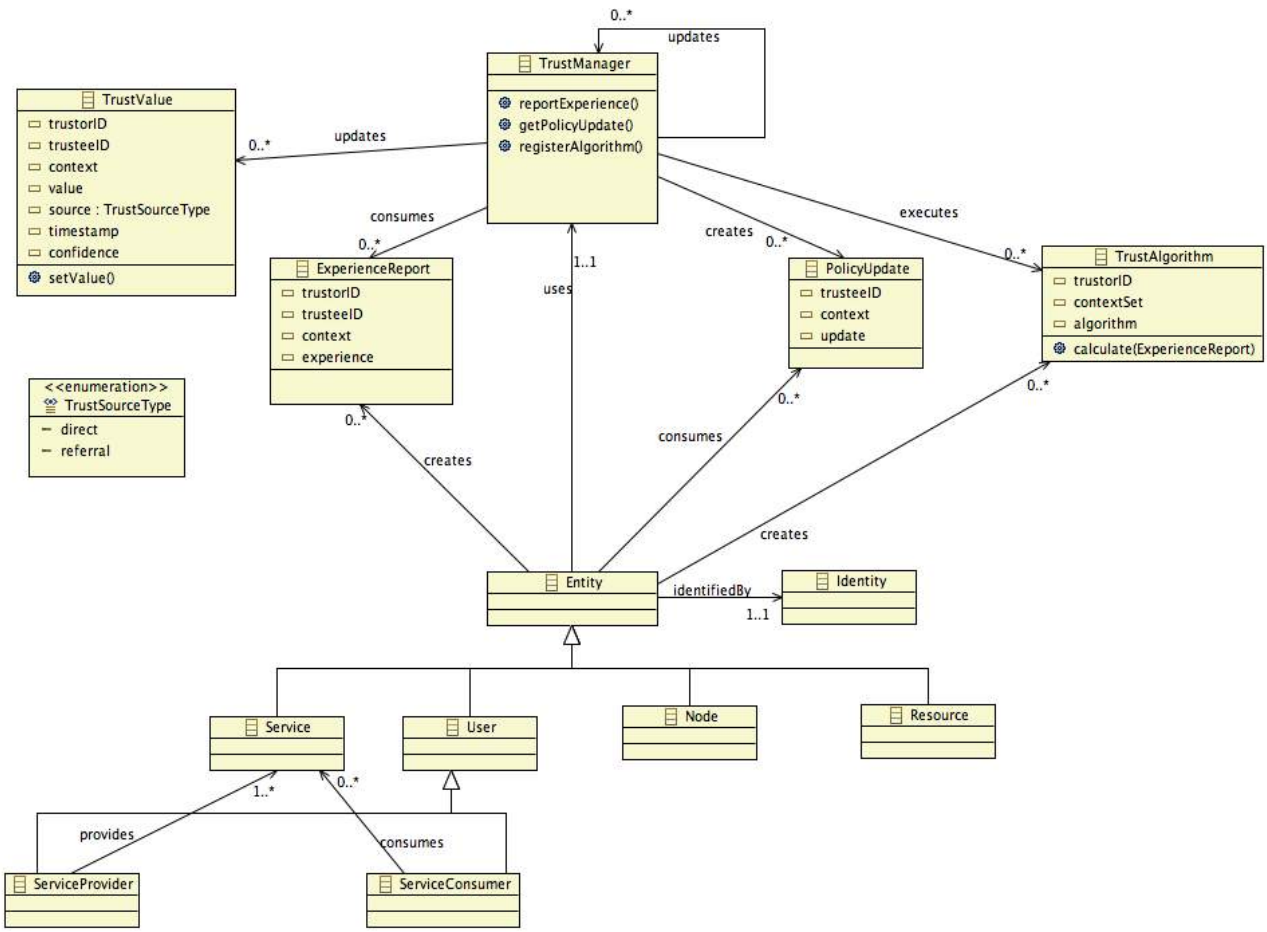


Figure 3.2: UML Class Diagram of TrustManager and associated classes

3.2 Trust Manager

The TrustManager acts as the core of the trust model. It has the following set of responsibilities:

1. Maintaining a set of algorithms for connected Trustor entities for specific contexts. Each context can have distinct algorithms for determining trust based on direct experience, referrals and reputation.
2. Receiving ExperienceReports from the Trustor entities and generating updated trust values according to the appropriate algorithms.
3. Providing updated TrustValues to other TrustManagers which can be used as inputs to referral or reputation algorithms. It is only appropriate that these published TrustValues have been derived from direct experience.
4. Providing PolicyUpdates to the Trustor entities based on current TrustValues derived from direct experience and referrals and reputation.

```
package org.opaals.trust;
public interface TrustManager {
        public boolean reportExperience(ExperienceReport experienceReport);
        public PolicyUpdate getPolicyUpdate(String trustorID,
                                            String trusteeID,
                                            Context context);
        public boolean registerAlgorithm(String trustorID,
                                            String context,
                                            String algorithm,
                                            TrustSourceType type);
        public void update(TrustValue update);
}
```

Figure 3.3: TrustManager interface

The TrustManager interface is shown in Figure 3.3. An initial implementation of this interface is provided in the code base and is called `org.opaals.trust.impl.SimpleTrustManger`. This implementation does not currently include any persistence of ExperienceReports or TrustValues. The next iteration of this implementation will make use of the Distributed Storage facilities of the underlying infrastructure as it becomes available.

3.3 Trust Values

A TrustValue represents a single instance of a trust value a relying party has in a trusted party in a particular context. An extract of the class is shown below in Figure 3.4. In effect each TrustValue instance is the output of a trust algorithm. The trust value itself is represented as float. Each TrustValue has a type associated with it. The type indicates the source of the data used to derive the value. Firstly, this can be direct experience, where the trust value was calculated using a direct

experience report. Secondly, this can be referral, where the relying party has received a trust value from a third party and has used an algorithm to calculate a trust value based on referrals. Thirdly, the trust value can be derived from a reputation based algorithm, where a set of trust values is used as an input. Finally, the trust value can be calculated using a hybrid of direct experience, referral and reputation algorithms. The enumerated class TrustSourceType is shown in Figure 3.4.

```
public class TrustValue {

        private String trustorID;
        private String trusteeID;
        private Context context;
        private float trustValue;
        private TrustSourceType trustType;
        private Date timestamp;
        private float confidence;
....
      // getter and setter methods
....
}

public enum TrustSourceType {

        DIRECT,
        REFERRAL,
        REPUTATION,
        HYBRID;
}
```

Figure 3.4: TrustValue class extract and TrustSourceType enumerated class

3.4 Experience Report

An ExperienceReport represents a unit of experience from an end user pertaining to an experience with a trustee in a context. The ExperienceReport forms the basis of all trust calculations as it is required to calculate any direct experience TrustValues. Referrals are only valid if their source is one of direct experience. The base ExperienceReport abstract class considers the experience as an Object. Each class extending ExperienceReport must implement `getExperience()` and `setExperience(Object experience)` as seen in Figure 3.5.

```
public abstract class ExperienceReport {

        private int id;
        private String trustorID;
        private String trusteeID;
        private Context context;
        private Object experience;
....

        public abstract Object getExperience();

        public abstract void setExperience(Object experience);

....

}
```

Figure 3.5: ExperienceReport abstract class extract

The creation of ExperienceReports is vital for the successful deployment of our trust network and two implementations are currently available. The first of these is called SimpleExperienceReport, which considers experience as a simple negative or positive value, i.e. the experience is either good or bad. The second implementation is called AccountedReport (see Figure 3.6) which makes use of data which will be created by our Accountability framework. It is a simple matter to implement other types of ExperienceReport, which are application dependent. This design allows for 3rd parties who may have experience capturing facilities in place to easily integrate into our trust framework. The ExperienceReport class itself does not have knowledge of the experience objects it handles. That is the role of the implementation of DirectExperienceTrustAlgorithm which has the responsibility of interpreting the experience as appropriate to trust value generation(see sections 3.5 and 3.6).

```
import org.opaals.accountability.usagedata.UsageData;

public class AccountedReport extends ExperienceReport {

        private UsageData experience;

        @Override
        public Object getExperience() {

                return experience;
        }

        @Override
        public void setExperience(Object experience) {

                this.experience = (UsageData)experience;
        }

}
```

Figure 3.6: AccountedReport

3.5 Trust Algorithms

Trust algorithms have the role of taking some known information and deriving a new trust value according to some predefined logic. A discussion of suitable trust algorithms is available in Deliverable D3.9. An interface for TrustAlgorithm is shown in Figure 3.7. In the base interface, two methods are defined. The first method, `setTrustValues(String, float)`, allows the user to input hard values for some trusted parties. For example, a service consumer might decide that it would like to overload the algorithm with some hardcoded values for some of the service providers it interacts with. The second method is `setParameters(String, String)` which is used to load parameters of the logic in the algorithm. An example of these parameters would be the rate of increase or decrease in trustworthiness for a specific context. Three further interfaces extend this base interface and these are discussed below.

3.5.1 DirectExperienceTrustAlgorithm

A direct experience algorithm is one which takes a representation of direct experience with a trusted party in a context and uses that experience to evaluate trustworthiness in that context. The resultant TrustValue may or may not be used to feed into the community based referral and reputation overlay. The interface DirectExperienceTrustAlgorithm extends the trustAlgorithm interface and defines one method, `execute(TrustValue, ExperienceReport)` that an implementing class must implemented (see Figure 3.7). The first parameter represents the current TrustValue for the trusted party in the current context. This can be null, in the case that no previous TrustValue exists. The second parameter is the ExperienceReport whose generation has caused this method to be called. An implementation of the exponential average direct experience algorithm discussed in Deliverable D3.9 is available at `org.opaals.trust.algorithm.impl`.

3.5.2 ReferralTrustAlgorithm

A referral trust algorithm is one which takes a referral from a third party and evaluates a TrustValue according to a predefined logic. The referral must be derived directly from direct experience as this is considered the only suitable input to a referral algorithm. The interface defines one method `execute(TrustValue, TrustValue)`, where the first parameter is the relying party's current TrustValue based on direct experience (see Figure 3.7). This can be null in the case where no direct experience is available. The second parameter is the referred TrustValue based on a (trusted in referrals) third party's direct experience. It should be noted that the means of evaluation of this referral (i.e. the DirectExperienceTrustAlgorithm implementation) is not known to the relying party. An implementation of the exponential average referral algorithm discussed in Deliverable D3.9 is available at `org.opaals.trust.algorithm.impl`.

3.5.3 ReputationTrustAlgorithm

A reputation trust algorithm can be viewed as a evaluation on a set of TrustValues. The trust values are a set of referrals for a trusted party in a particular context. The ReputationTrustAlgorithm interface extends TrustAlgorithm defines a method `execute(TrustValue current, TrustValueList referrals)`(see Figure 3.7). The first parameter is the relying party's current TrustValue for the trusted party. The second parameter is a set of TrustedValues of referrals from third parties' direct experiences with the trusted party in the same context.

```
package org.opaals.trust.algorithm;

public interface TrustAlgorithm {
	public void setTrustValues(String trustee, float value);
	public void setParameters(String name, String value);
}

public interface DirectExperienceTrustAlgorithm extends TrustAlgorithm{

	TrustValue execute(TrustValue current, ExperienceReport experienceReport);

}

public interface ReferralTrustAlgorithm extends TrustAlgorithm{

	TrustValue execute(TrustValue directValue, TrustValue referredValue);

}

public interface ReputationTrustAlgorithm extends TrustAlgorithm{

	TrustValue execute(TrustValue current, TrustValueList referrals);

}
```

Figure 3.7: TrustAlgorithm interfaces

3.6 Configuring Algorithms

Each TrustManager manages a set of trust algorithms for various contexts. As each relying party has a TrustManager to manage its preferred algorithms, it is useful to devise a means of configuring the TrustManager to achieve this on a run time basis. To aid this a TrustAlgorithmConfiguration XML schema is designed (see Appendix A for details of the schema). Trust Contexts are configured separately and each Context can define TrustAlgorthms for direct experience, referrals and reputation. An example XML instance of this schema is shown in Figure 3.8. In the example two contexts are configured. The first Context is called 'availability' and shows how a direct experience algorithm is configured for that context. One parameter value is also configured and also one trust value. The second context is called 'consistency' and shows how a referral algorithm is configured.

```
<?xml version="1.0" encoding="UTF-8"?>
<config:trustAlgorithmConfiguration xmlns:config="http://www.opaals.org/TrustAlgorithmConfig">

  <config:context name="availability">
    <config:contextImpl>org.opaals.trust.impl.SimpleContext</config:contextImpl>
    <config:trustAlgorithm weight="1">
      <config:type>DirectExperience</config:type>
      <config:implementation>org.opaals.trust.algorithm.impl.EasyAlgo</config:implementation>
      <config:initialValue>0</config:initialValue>
      <config:values>
        <config:trustValue>
          <config:trusteeID>google.com</config:trusteeID>
          <config:value>0.99</config:value>
        </config:trustValue>
      </config:values>
      <config:variables>
        <config:variable>
          <config:name>alpha</config:name>
          <config:value>0.3</config:value>
        </config:variable>
        <config:variable>
          <config:name>beta</config:name>
          <config:value>0.03</config:value>
        </config:variable>
      </config:variables>
    </config:trustAlgorithm>
  </config:context>

  <config:context name="consistency">
    <config:contextImpl>org.opaals.trust.impl.SimpleContext</config:contextImpl>
    <config:trustAlgorithm weight="0.5">
      <config:type>Referral</config:type>
      <config:implementation>org.opaals.trust.algorithm.impl.EasyAlgoRef</config:implementation>
      <config:initialValue>0</config:initialValue>
      <config:values>
        <config:trustValue>
          <config:trusteeID>gmail.com</config:trusteeID>
          <config:value>0.9</config:value>
        </config:trustValue>
      </config:values>
      <config:variables>
        <config:variable>
          <config:name>omega</config:name>
          <config:value>0.3</config:value>
        </config:variable>
        <config:variable>
          <config:name>delta</config:name>
          <config:value>0.03</config:value>
        </config:variable>
      </config:variables>
    </config:trustAlgorithm>
  </config:context>
</config:trustAlgorithmConfiguration>
```

Figure 3.8: Trust Algorithm Configuration XML

3.7 A simple example

A simple example of how the TrustManager and TrustAlgorithms can be used to determine evolve trust is provided in org.opaals.trust.example.SimpleClient. The example makes use of the ExponentialAverageDirectExperienceTrustAlgorithm. The execute method for this algorithm is shown in Figure 3.9.

```
public class ExponentialAverageDirectExperienceTrustAlgorithm implements
DirectExperienceTrustAlgorithm {
	....
	public TrustValue execute(TrustValue trustValue,
			ExperienceReport experienceReport) {

		float currentTrust = trustValue.getTrustValue();
		float experience;

		SimpleExperienceReport simpleExperienceReport = (SimpleExperienceReport)
experienceReport;
		if ( (SimpleExperience)simpleExperienceReport.getExperience() ==
SimpleExperience.POSITIVE)
			experience = 1;
		else
			experience = 0;

		float newTrust = (alpha * experience) + ((1 - alpha) * currentTrust);

		trustValue.setTrustValue(newTrust);
		trustValue.setTimestamp(new Date());

		return trustValue;
	}
```

Figure 3.9: The ExponentialAverageDirectExperienceTrustAlgorithm exectute() method

The example client program creates SimpleExperienceReports and reports these experiences to a TrustManager instance which has been configured to use the ExponentialAverageDirectExperienceTrustAlgorithm for the context of availability. One hundred reports are created an a behaviour is emulated to show the effect of positive and negative experiences have on the trust output of this algorithm. The behaviour is show in Figure 3.10. Positive experience is reported for the first 50 emulations. One negative experience is reported following this. The experience then becomes positive again for 10 emulations. There then follows 30 negative experiences briefly interrupted with 1 positive experience. The final 10 experiences are positive. The algorithm uses a parameter called alpha which controls the rate of change of the trust. The higher the value of alpha, the faster trust evolves towards 1 for positive experience and towards 0 for negative experience. Two outputs of this program are shown in Figures 3.12 and 3.11,

one for alpha equal to 0.05 and one for alpha equal to 0.15.

```
SimpleExperienceReport ser = new SimpleExperienceReport();
ser.setTrustorID(trustorID);
ser.setTrusteeID(trusteeID);
ser.setContext(context);
for(int i = 0; i < numOfExperiences; i++){

        if (i % 50 == 0)
        ser.setExperience(SimpleExperience.NEGATIVE);
        else if (i % 69 == 0)
                ser.setExperience(SimpleExperience.POSITIVE);

        else if (i > 60 && i < 89)
                ser.setExperience(SimpleExperience.NEGATIVE);

        else
                ser.setExperience(SimpleExperience.POSITIVE);

        trustManager.reportExperience(ser);

        PolicyUpdate policy = trustManager.getPolicyUpdate(trustorID, trusteeID, context);
        TrustValue val = policy.getUpdate();
}
```

Figure 3.10: How the behaviour is modelled in the sample client code

Figure 3.11: Trust Evolution for Exponential Average Direct Experience Trust Algorithm (alpha = 0.05)

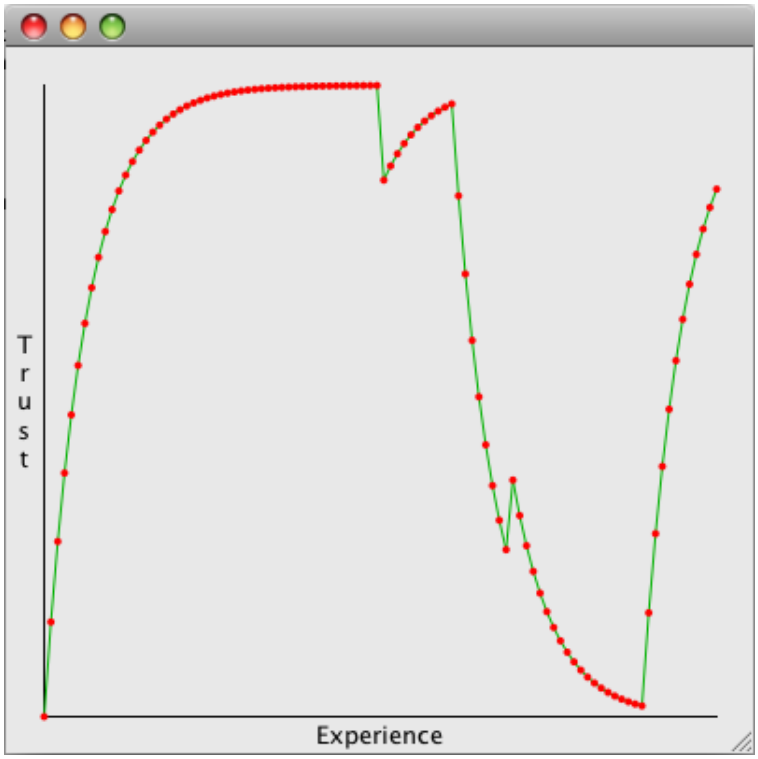


Figure 3.12: Trust Evolution for Exponential Average Direct Experience Trust Algorithm (alpha = 0.15)

3.8 Open Source Project – TrustFlow

A project has been established on sourceforge to house the development of the OPAALS trust model. The project is available at http://trustflow.sourceforge.net and contains the source code as well as standalone examples showing how the framework is used. Sample trust algorithms are provided.

4 Distributed Accountability Implementation

The OPAALS distributed accountability model is described in deliverable D3.8. This model is shown below in Figure 4.1. Although the diagram shows a simple transaction between two services, The introduction of the *Accounting Authority* provides the required functionality necessary for composed services. The role of the *Accounting Authority* is to ensure accountability for composed services can be provided by the super-set of all mediation peers involved in the service composition.

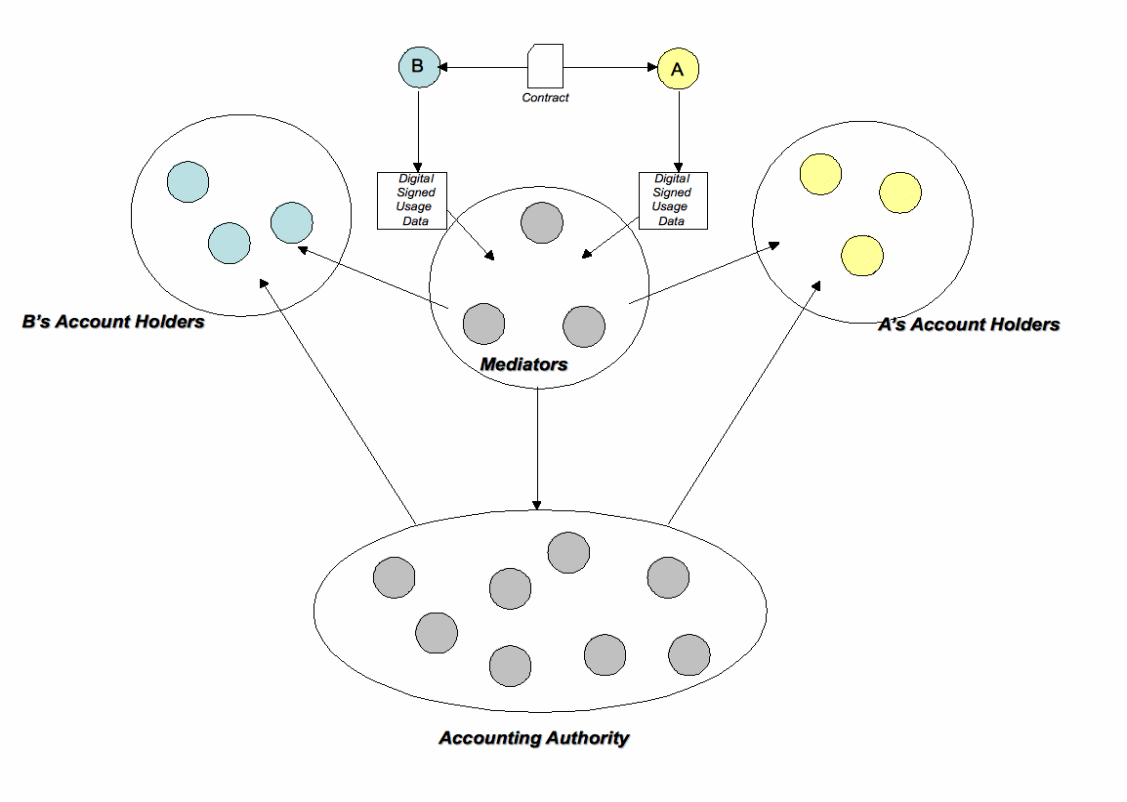


Figure 4.1: Distributed Accountability Model

A UML class diagram of the actors and the protocol is shown in Figure 4.2. The interfaces of this package are shown in the following section. All peers who participate in accountable services must use an implementation of an AccountableEntity. The AccountableEntity implements methods necessary for the protocol to be instantiated, including the Mediator, AccountHolder and AccountingAuthorityMember interfaces.

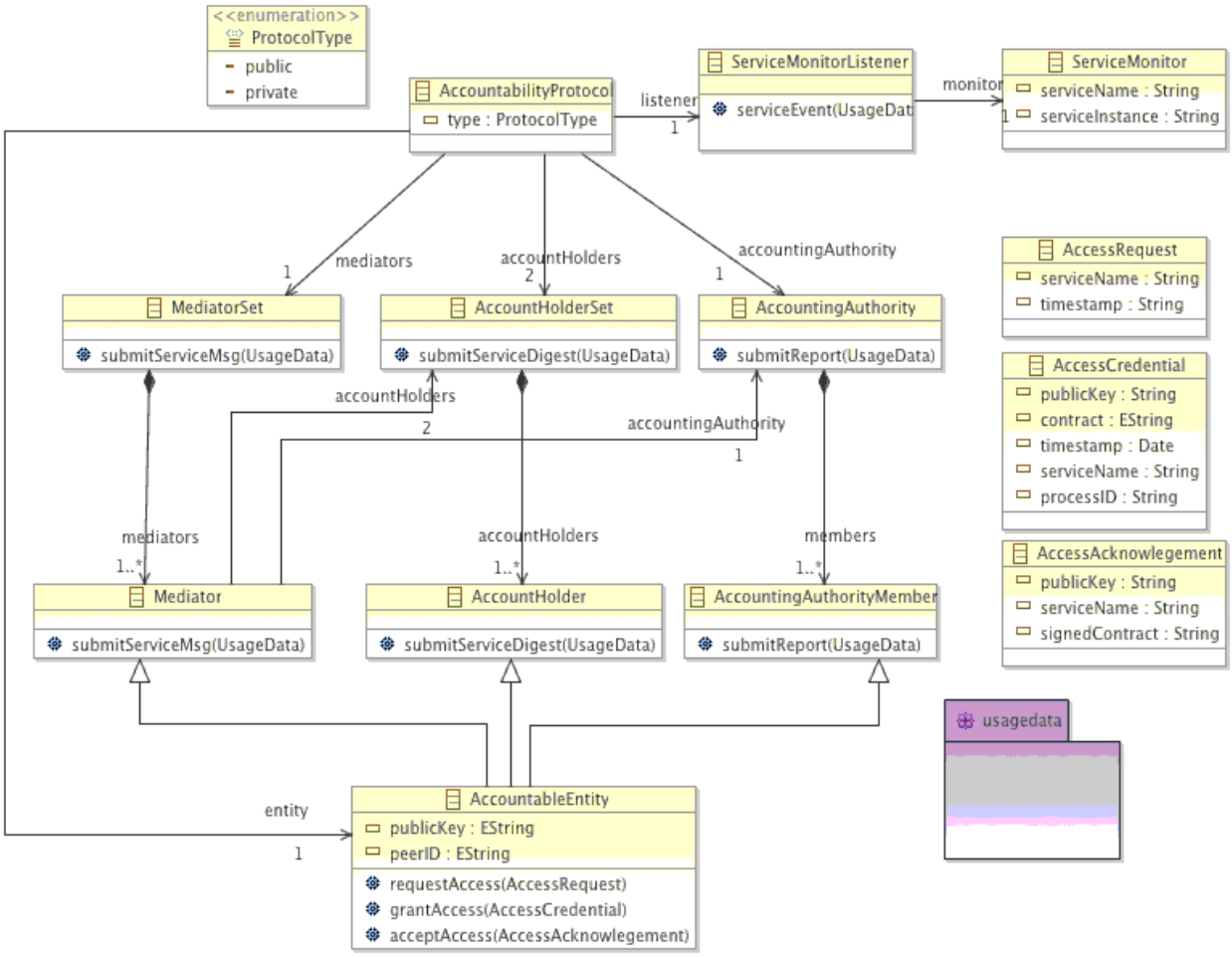


Figure 4.2: UML Class diagram of org.opaals.accountability

4.1 The Actors

Peers

The Peers are responsible for the creation of the evidence and the delivery to the Mediators. Peers can also act as Mediators, Account Holders and members of Accounting Authorities.

Mediators

The Role of the Mediators is to collect the evidence from each Peer and compare both records for consistency. Alarms are raised when disagreement is discovered. Mediators also release periodic digests to Account Holders and Accounting Authorities. Selected Peers take on the Mediator role for the duration of the session or transaction.

Account Holders

The Account Holders are responsible for the delivery, persistence and retrieval of accounted data.

Account Holders are more long-lived than Mediators and Peers are periodically reselected through the issue of a nonce by the Peer for which they persist data.

Accounting Authorities

An Accounting Authority delivers service composition relative data to Account Holders for persistence as well as raising alarms in the case of service composition inconsistencies. Peers operate as Accounting Authorities when they are Mediators and Peers of a service composition scenario.

The interfaces which define these actors for implementation are outlined below

4.1.1 The Mediator interface

The Mediator is responsible for receiving accounted data for service usage in real time. The mediator is also responsible for validating the integrity of this usage data and submitting reports of this data to the Accounting Authority and ultimately providing a digest of the complete service usage history to the AccountHolders of the AccountableEntities. The interface is shown in Figure 4.3.

```
package org.opaals.accountability;

import org.opaals.accountability.usagedata.UsageData;

public interface Mediator extends Object {
	List<AccountHolderSet> getAccountHolders();
	AccountingAuthority getAccountingAuthority();
	void setAccountingAuthority(AccountingAuthority value);
	void submitServiceMsg(UsageData message);
}
```

Figure 4.3: The Mediator interface

4.1.2 The AccountHolder interface

The AccountHolder interface specifies one method specifically for participation in accountability protocols. Further methods need to be specified in order for peers to query accounted data for transactions in which they have participated. This will be specified in the next phase of the project.

```
package org.opaals.accountability;

import org.opaals.accountability.usagedata.UsageData;

public interface AccountHolder extends Object {

	void submitServiceDigest(UsageData digest);

}
```

Figure 4.4: The AccountHolder interface

4.1.3 The Accounting Authority interface

The AccountingAuthority checks the integrity of the accounted UsageData within the scope of a transaction, in the case of composed services using the OPAALS distributed transaction model.

```
package org.opaals.accountability;

import org.opaals.accountability.usagedata.UsageData;

public interface AccountAuthority extends EObject {

	EList<AccountingAuthorityMember> getMembers();
	void submitReport(UsageData report);

}
```

Figure 4.5: The AccountAuthority interface

4.1.4 The AccountableEntity interface

The AccountableEntity interface represents the Peer actor and extends the Mediator, AccountHolder and AccountingAuthorityMember interfaces. Methods to aid in the correct running of the protocol are also specified. Each peer wishing to participate in accountable service consumption or provision must provide an implementation of this interface to the Protocol. An implementation of this interface for FlyPeer services will be supplied in the next phase of the project.

```
package org.opaals.accountability;

public interface AccountableEntity extends Mediator, AccountHolder, AccountingAuthorityMember
{
	String getPublicKey();
	void setPublicKey(String value);
	String getPeerID();
	void setPeerID(String value);
	void requestAccess(AccessRequest request);
	void grantAccess(AccessCredential credential);
	void acceptAccess(AccessAcknowlegement ack);

}
```

Figure 4.6: The AccountableEntity interface

4.2 Protocols

Two protocols are catered for, one for accountability of public data and one for private data exchange. The protocols are similar except that in the case of the private accountability the messages are encrypted with a shared secret. The protocols are discussed in detail in Deliverable D3.8.

4.2.1 The AccountabilityProtocol interface

The AccountabilityProtocol interface (shown in Figure 4.7) is responsible for the overall control of the data flow and the coordination of the protocol. The AccountabilityProtocol works on behalf of an Accountable Entity. The type (public or private) is set by the setType() method. The UsageData is collected from the ServiceMonitorListener and sent to the MediatorSet via the submitServiceMsg(UsageData) method.

```
package org.opaals.accountability;

public interface AccountabilityProtocol extends Object {
        MediatorSet getMediators();
        EList<AccountHolderSet> getAccountHolders();
        AccountingAuthority getAccountingAuthority();
        ProtocolType getType();
        ServiceMonitorListener getListener();
        AccountableEntity getEntity();

        void setMediators(MediatorSet mediators);
        void setAccountingAuthority(AccountingAuthority accountingAuthority);
        void setType(ProtocolType type);
        void setListener(ServiceMonitorListener listener);
        void setEntity(AccountableEntity entity);
}
```

Figure 4.7: The AccountabilityProtocol interface

4.3 Usage Data

An XML Schema to record the usage data allowing for signing and enciphering of data is shown below in Figure 4.8. Each instance of a UsageData document contains a UsageHeader and a number of UsageRecords. The UsageHeader contains information about the provider and consumer of the service as well as the serviceID, the processID and the time stamp when the transaction began. Each UsageRecord contains an attribute indicating which protocol is being used, the ID of the initiator (who generated the record), the encrypted messageData, the initiator's signature and a time stamp of when the data was metered.

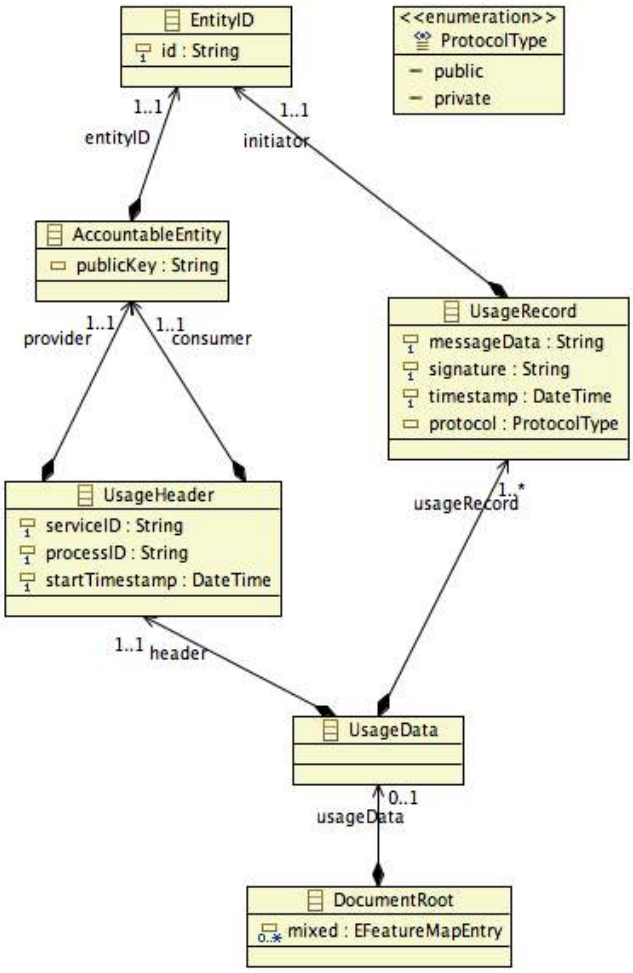


Figure 4.8: UML Class diagram of org.opaals.accountability.usagedata

5 Conclusion and Next Steps

This document has provided an update of progress on the implementation of distributed identity, trust and accountability in OPAALS. The work on identity is complete and only remains that this is integrated successfully into the FlyPeer platform. The trust model implementation is progressing well and requires some further work with the development of algorithms and integration of the experience reports with usage data from the accountability framework. The initial work on Accountability is in place and this work will progress further in the coming months.

Deliverable D5.5 will provide a comprehensive report on the integration of these components with the FlyPeer platform.

Appendix A – Trust Algorithm Configuration XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.opaals.org/TrustAlgorithmConfig"
xmlns:tns="http://www.opaals.org/TrustAlgorithmConfig" elementFormDefault="qualified">

    <element name="trustAlgorithmConfiguration" type="tns:trustAlgorithmConfigurationType"/>

    <complexType name="contextType">
       <sequence>
              <element name="contextImpl" type="string" maxOccurs="1" minOccurs="1"/>
              <element name="trustAlgorithm" type="tns:trustAlgorithmType" minOccurs="1"
maxOccurs="3"></element>
       </sequence>
       <attribute name="name" type="string"></attribute>
    </complexType>

    <complexType name="trustAlgorithmType">
       <sequence>
              <element name="type" type="tns:trustAlgorithmSourceType" maxOccurs="1"
minOccurs="1"/>
              <element name="implementation" type="string" maxOccurs="1" minOccurs="1"/>
              <element name="initialValue" type="string" maxOccurs="1" minOccurs="0"/>
              <element name="values" type="tns:staticTrustValueSetType" maxOccurs="1"
minOccurs="0"></element>
              <element name="variables" type="tns:variableSetType" maxOccurs="1"
minOccurs="0"></element>
       </sequence>
       <attribute name="weight" type="string"></attribute>
    </complexType>

    <simpleType name="trustAlgorithmSourceType">
           <restriction base="normalizedString">
               <enumeration value="DirectExperience"/>
               <enumeration value="Referral"/>
               <enumeration value="Reputation"/>
           </restriction>
       </simpleType>

    <complexType name="variableType">
       <sequence>
              <element name="name" type="string"></element>
              <element name="value" type="string"></element>
       </sequence>
    </complexType>

    <complexType name="variableSetType">
       <sequence>
              <element name="variable" type="tns:variableType" minOccurs="0"
maxOccurs="unbounded"></element>
       </sequence>
    </complexType>

    <complexType name="staticTrustValueType">
       <sequence minOccurs="1" maxOccurs="1">
              <element name="trusteeID" type="string"></element>
              <element name="value" type="string"></element>
       </sequence>
    </complexType>

    <complexType name="staticTrustValueSetType">
       <sequence>
              <element name="trustValue" type="tns:staticTrustValueType" minOccurs="0"
maxOccurs="unbounded"></element>
       </sequence>
    </complexType>

    <complexType name="trustAlgorithmConfigurationType">
```

```
        <sequence>
                <element name="context" type="tns:contextType"></element>
        </sequence>
    </complexType>
</schema>
```

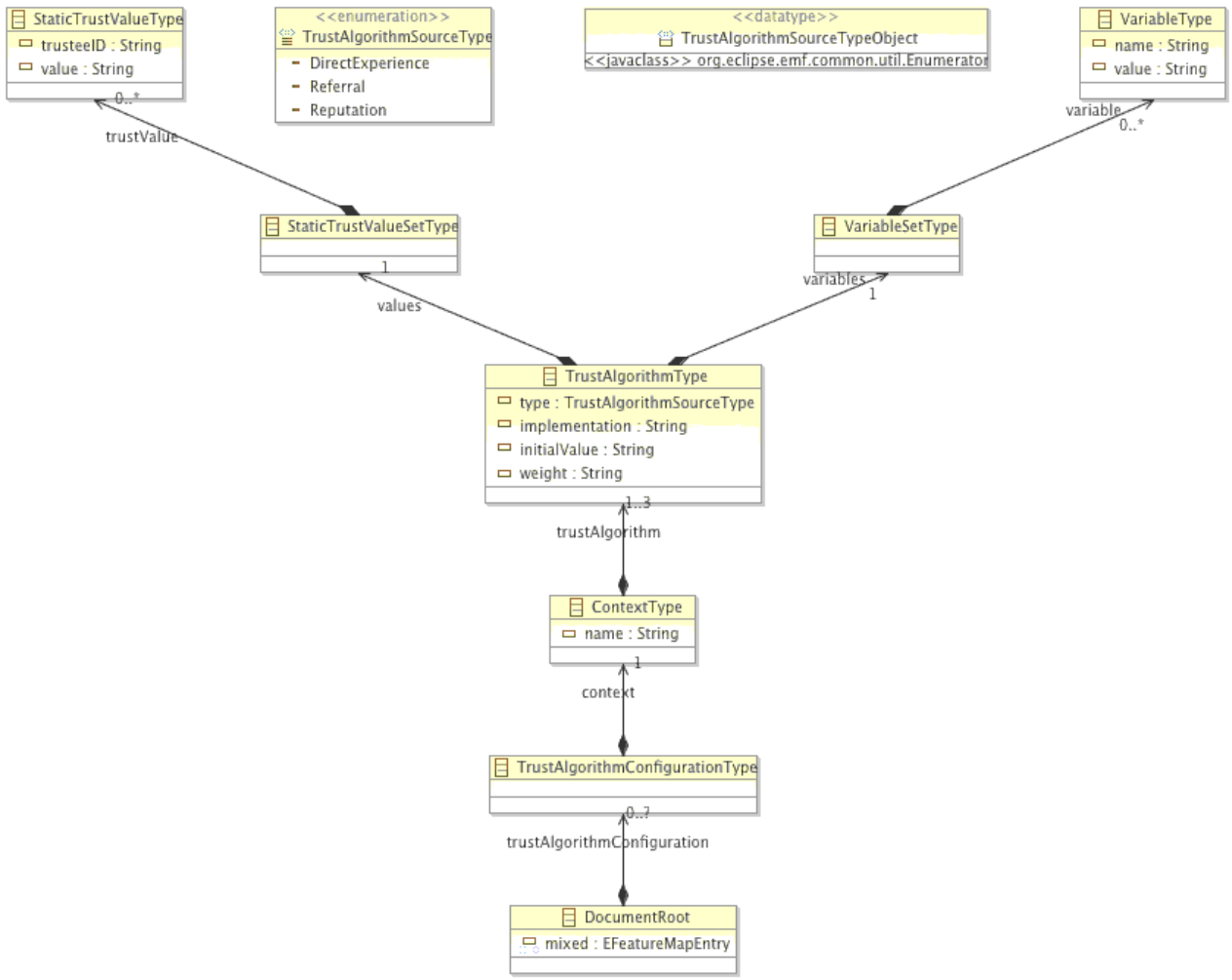


Figure 1: Trust Algorithm Configuration Model

Appendix B – Usage Data XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.opaals.org/schema/usagedata"
    xmlns:tns="http://www.opaals.org/schema/usagedata"
    elementFormDefault="qualified">
    <xsd:element name="usagaeData">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="header" type="tns:usageHeaderType" minOccurs="1" maxOccurs="1"/>
                <xsd:element name="usageRecord" type="tns:usageRecordType" minOccurs="1"
maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="usageHeaderType">
        <xsd:sequence>
            <xsd:element name="consumer" type="tns:accountableEntityType" minOccurs="1"
maxOccurs="1"></xsd:element>
            <xsd:element name="provider" type="tns:accountableEntityType" minOccurs="1"
maxOccurs="1"></xsd:element>
            <xsd:element name="serviceID" type="xsd:string" minOccurs="1"
maxOccurs="1"></xsd:element>
            <xsd:element name="processID" type="xsd:string" minOccurs="1"
maxOccurs="1"></xsd:element>
            <xsd:element name="startTimestamp" type="xsd:dateTime" minOccurs="1"
maxOccurs="1"></xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="entityIDType">
        <xsd:sequence>
            <xsd:element name="id" type="xsd:string" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="accountableEntityType">
        <xsd:sequence>
            <xsd:element name="entityID" type="tns:entityIDType"></xsd:element>
            <xsd:element name="publicKey" type="xsd:string"></xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="usageRecordType">
        <xsd:sequence>
            <xsd:element name="initiator" type="tns:entityIDType"/>
            <xsd:element name="messageData" type="xsd:string"/>
            <xsd:element name="signature" type="xsd:string"/>
            <xsd:element name="timestamp" type="xsd:dateTime"/>
        </xsd:sequence>
        <xsd:attribute name="protocol">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="public"/>
                    <xsd:enumeration value="private"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
</xsd:schema>
```