SmartSociety

Hybrid and Diversity-Aware Collective Adaptive Systems
When People Meet Machines to Build a Smarter Society

Grant Agreement No. 600584

Deliverable 8.4 Working Package 8

Final platform prototype and validation

Dissemination Level (Confidentiality):[1]	*PU*
Delivery Date in Annex I:	*30/06/2016*
Actual Delivery Date	*August 9, 2016*
Status[2]	*D*
Total Number of pages:	48
Keywords:	*Platform, Prototype, Integration, Validation*

[1]PU: Public; RE: Restricted to Group; PP: Restricted to Programme; CO: Consortium Confidential as specified in the Grant Agreeement
[2]F: Final; D: Draft; RD: Revised Draft

Disclaimer

Full project title:	SmartSociety: Hybrid and Diversity-Aware Collective Adaptive Systems: When People Meet Machines to Build a Smarter Society
Project Acronym:	SmartSociety
Grant Agreement Number:	600854
Number and title of work-package:	8 Architecture and Integration
Document title:	Final platform prototype and validation
Work-package leader:	Iacopo Carreras, UH
Deliverable owner:	Daniele Miorandi, UH
Quality Assessor:	Luc Moreau, SOTON

List of Contributors

Partner Acronym	Contributor
UH	Tommaso Schiavinotto, Daniele Miorandi, Iacopo Carreras

Executive Summary

The SmartSociety platform represents a toolbox for easily and quickly building applications and services based upon hybrid, diversity-aware forms of social computation. The toolbox includes eight components, each one performing a well-defined functionality and accessible through REST APIs. Functionality exposed by the components can be accessed by means of a set of programmer-friendly primitives, which are supported by a purposefully developed runtime library. Java bindings are provided as part of the toolbox.

The core components of the toolbox are released as open source form under a liberal licensing scheme, allowing researchers, Web entrepreneurs and startup companies to quickly prototype, deploy and experiment with applications and services enabling augmented collectives.

This report describes the final version of the SmartSociety platform, which integrates the instantiation of the components designed and developed by the Consortium members within the technical workpackages WP2-WP7.

Table of Contents

List of Acronyms

Acronym	Full Name	Description
CAS	Collective Adaptive System	
HDA-CAS	Hybrid and Diversity-Aware Collective Adaptive System	
API	Application Programming Interface	Set of tools and methods for accessing, from within a computer program, the functionality of an application/service
REST	Representational State Transfer	An architectural style of APIs, where the focus is on component roles and a specific set of interactions between data elements rather than implementation details.
CM	Context Manager	System component in charge of monitoring the context the agent represented on the platform by a peer is in.
IM	Incentives Manager	System component in charge of managing the implementation of incentive schemes.
KB	Knowledge Base	System component in charge of storing and managing the knowledge in the platform.
OM	Orchestration Manager	System component in charge of orchestrating the lifecycle of tasks on the SmartSociety platform.
PF	Programming Framework	System component in charge of exposing appropriate primitives and interfaces to application developers.
PM	Peer Manager	System component in charge of managing peers.
PS	Provenance Service	System component in charge of logging actions and information flows occurring on the platform.
RM	Reputation Manager	System component in charge of handling the reputation of any system resource, including peers.
SmartCom	Communication Middleware	System component in charge of managing communication channels between the platform and the peers.
TEM	Task Execution Manager	System component in charge of monitoring the execution of tasks and trigger corrective actions if needed.

1 Introduction

This report describes the final version of the SmartSociety platform. The corresponding software toolbox can be accessed at the following integrated repository: `https://gitlab.com/smartsociety/`

The SmartSociety platform integrates the technical components, and provides the hooks to be able to operate a HDA-CAS. Generally speaking, a HDA-CAS is composed by peers (individuals, machines and collectives representing ensembles of the aforementioned) which coalesce around (and is enabled by) a service/application, which may run on top of the SmartSociety platform. The platform itself is therefore a technology enabler for the service/application which gives rise to the HDA-CAS.

From the technical standpoint, the platform is shipped as a toolbox, and it includes eight components, each one performing a well-defined functionality and accessible through REST APIs. Functionality exposed by the components can be accessed directly or by means of a set of programmer-friendly primitives, supported by a purposefully developed runtime library. Java bindings are provided as part of the toolbox.

The core components of the toolbox are released as open source form under a liberal licensing scheme (Apache v.2[3]). This is in line with the mission of the SmartSociety, where the platform plays a pivotal role in allowing researchers, Web entrepreneurs and startup companies to quickly prototype, deploy and experiment with applications and services based upon hybrid and diversity-aware collectives.

The final version of the platform builds upon v.2.0, which was delivered as D8.2 [2]. The enhancements include:

- Integration of the Task Execution Manager (TEM), the component in charge of monitoring the execution of tasks and identify deviations from the plan (and, in case, call the Orchestration Manager to provide a new plan);
- Integration of the Incentive Server (IS), the component providing on-line recommendations on how to improve peer engagement;
- Integration of the programming framework (PF), which provides primitives for programmers to easily access the platform functionalty.

[3] http://www.apache.org/licenses/LICENSE-2.0

The remainder of the deliverable is organized as follows. Sec. 2 describes the platform architecture and the overall functionality. Sec. 3 describes the platform components and their interfaces. Sec. 4 describes how the platform can be used to build CASs and includes some basic examples. Sec. 5 analyses the ability of the platform in its final version to meet the requirements elicited in [1]. Sec. 6 concludes the report with an outlook on the future of the platform and its potential exploitation, together with some ethics considerations on the usage of the platform.

2 Platform Architecture

A high-level view of the SmartSociety platform architecture is presented in Fig. 1 [3]. The rectangle boxes represent the key platform components. All components expose RESTful APIs. The deployment can be centralized or distributed, depending on the specific use cases and constraints.

In general, we distinguish between the following relevant actors [9]:

1. Users external human clients or applications who require a task to be performed;
2. Peers human or machine entities providing Human/Web Services to the platform;
3. Developers external individuals providing the business logic in form of a programming language code that is compiled and executed on the platform in form of SmartSociety platform applications.

In the SmartSociety framework peers typically perform tasks within collectives. A collective consists of a team of human and/or machine peers, whose formation and operations are purposefully supported by the platform.

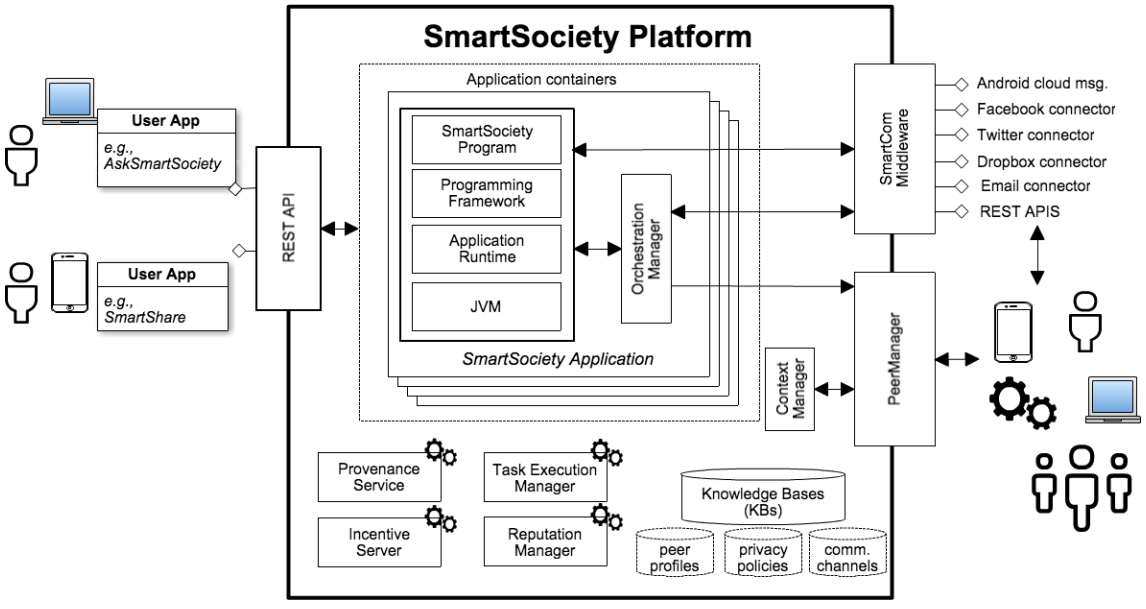

Figure 1: SmartSociety platform users and architecture.

From the logical standpoint, the platform includes eight components, each one performing a well-defined functionality and accessible through REST APIs. Functionality

exposed by the components can be accessed by means of a set of programmer-friendly primitives (through the so-called Programming Framework), which are supported by a purposefully developed runtime library. Java bindings are provided as part of the toolbox. The platform components are:

- **Orchestration Manager (OM)**: it provides two key functionalities: composition and negotiation [4]. Composition takes task requests as inputs and generates tasks by solving constraint satisfaction problems specific to each application, potentially interacting with the peer manager (so that peer with certain capabilities can be identified) and the reputation service (so that the reputation of the capable peers can be obtained). The composition determines (i) the peers which can potentially execute such task, and (ii) the necessary interactions and/or external services which are needed to execute such task. The negotiation manager is in charge of handling the negotiation process with peers in order to ensure that the services and resources required to carry out the task are actually present.

- **Peer Manager (PM)**: it is responsible for managing peer-related information [5]. This includes their profile, as well as any other information which will be used by the Orchestration Manager for identifying possible peers that can execute a given task. It maintains a profile of each peer, which represents a model thereof in terms of knowledge, resources and capacity. It provides a peer search functionality that is used by the Orchestration Manager to create compositions. Besides individual peers, the Peer Manager also manages collectives, which are groups of peers characterized by specific properties. The Peer Manager includes a set of mechanisms for ensuring the user's privacy [5].

- **Task Execution Manager (TEM):** The TEM is in charge of managing the task execution phase. In particular it monitors whether the task execution is in line with expected plan (taken from the task description). In case a deviation is detected, it might trigger a re-planning by using OM's functionality.

- **Context Manager (CM):** it is responsible for monitoring the context the agent represented on the platform by a peer is in [6]. For an individual human agent, this includes, e.g., tracking the location and recognizing the activity the agent is currently involved in. This can be further used to trigger context adaptation in applications. The

- **Reputation Service (RS):**: it handles the reputation of any system resource, including peers [7]. It computes the reputation of a given peer based on feedback from other peers. Reputation information can be exposed directly to application users or used by the PM in the selection of suitable peers for a given task.
- **Provenance Service (PROV):** it logs descriptions of actions performed by platform components and peers, as well as information flows between them, according to the W3C PROV recommendation [8]. In particular, it includes the provenance store, a component responsible for keeping a record of how the overall compositions are being created, executed and how the data managed by the platform is being transformed.
- **Monitoring and analysis service (MONITOR):** it logs and monitors the platform jobs and can be used by platform administrators to check the liveness of the platform services, to perform root-cause analysis and to extract analytics on the performance of the system [**?**].
- **Incentives server (IS):** it manages incentives and interventions that can be used to achieve higher quality results. Incentives can be used to stimulate the participation of a peer (or a collective) in the execution of a given task or can be used to define specific strategies for community engagement [**?**].

3 Components and APIs

In this section we describe the software artifacts being produced, together with a description of their APIs.

3.1 Orchestration Manager

3.1.1 Functionality and Features

The OM provides collective composition and negotiation functionality. They can be invoked subsequently or jointly (in the latter case we talk about continuous orchestration, see [4] for a detailed description). The OM takes as input task requests and outputs an agreed execution plan.

3.1.2 Implementation

The OM implementation shipped with the toolbox is written in `JavaScript` and based upon the `node.js` framework. It uses `express` for REST APIs, `jade` as node template and includes a `MongoDB` instance for persistency. Two versions are provided, supporting AskSmartSociety! and SmartShare applications. They can be easily reused to develop application-specific OMs.

3.1.3 Interfaces, Endpoints and Resources Exposed

@Zhenyu/Michael: please check

We start with task requests:

- **POST /applications/:app/taskRequests** This is the main URI where new task requests are posted. The JSON object describing the task request is expected in the body of the request. On success a platform call to the composition manager will be made.
- **GET /applications/:app/taskRequests/?user=:user** Get all task requests posted by a given user.
- **GET /applications/:app/taskRequests/:taskRequestID** Get details on a given task request.

- **HEAD /applications/:app/taskRequests/:taskRequestID** Get the head of a given task request.
- **GET /applications/:app/taskRequests/:taskRequestID/v/:version** Get a specific version of a given task request.
- **DELETE /applications/:app/taskRequests/:taskRequestID** Delete a task request.

Tasks are generated through composition. The most basic operations related to them are listed in the following table:

- **GET /applications/:app/tasks/:taskID** Get a specific task (latest version).
- **GET /applications/:app/tasks/:taskID** Get the head of a specific task.
- **GET /applications/:app/tasks/:taskID/v/:version** Get a specific version of a given task.
- **PUT applications/:app/tasks/:taskID** Negotiate on a task. The main call for negotiation which will trigger an additional platform call to the negotiation manager. Expects the new version of the document of the task taskID. A platform job for negotiation is prepared and is posted to the negotiation manager.

Task records are generated by the orchestrator once execution can start on a specific task. The most basic operations are listed in the following table:

- **GET /applications/:app/taskRecords/:taskRecordID** Get a specific task record.
- **HEAD /applications/:app/taskRecords/:taskRecordID** Get the head of specific task record. This is another convenience function which allows an easy way for the clients to figure out if the resource has changed.
- **GET /applications/:app/taskRecords/:taskRecordID/v/:version** Get a specific version of a given task record.
- **PUT /applications/:app/taskRecords/:taskRecordID** Provide execution feebdack. The main call for execu- tion which will trigger an additional platform call to the execution manager. Expects the new version of the task record document taskRecordID. A platform job for execution is prepared and is posted to the execution manager.

The composition manager provides the following functionality:

- **POST /applications/:app/compositions** Perform composition. Expects the platform job with the description, for which the main ingredient is the new task request that has arrived on the platform. The orchestrator for the application app can make such a call.
- **GET /applications/:app/compositions/:compositionID** Get composition results. Normally such a call is expected to happen only once from the application orchestrator once the latter has received the 201 error code that the composition that was requested has been performed.

The negotiation manager provides the following functionality:

- **POST /applications/:app/negotiations** Perform negotiation. Expects the platform job with the description, for which the main ingredient is the task on which negotiation is being performed. Access Control. The call always succeeds generates a resource describing the outcome of negotiation. Upon completion it returns an error code 201 and the link to the document with the results of the negotiation. Part of the description of the document with the results of the negotiation is the error code and message that is returned through the call **PUT /applications/:app/tasks/:taskID** to the client.
- **GET /applications/:app/negotiations/:negotiationID** Get negotiation results. Normally such a call is expected to happen only once from the application orchestrator once the latter has received the 201 error code that the negotiation that was requested has been performed.

3.1.4 Repository

The OM code is available (Apache v.2) at: `https://gitlab.com/smartsociety/orchestration`

3.2 Peer Manager

3.2.1 Functionality and Features

The Peer Manager (PM) provides a peer-centered data store that maintains and manages information about human- or machine-based peers within a privacy-preserving framework.

The PM was designed with the objective of keeping the information owned by peers private. In order to provide a flexible management of information, the PM builds upon the notion of an entity-centric semantic enhanced model that defines an extensible set of entity schemas providing the templates for an attribute-based representation of peers characteristics [11]. Additionally, the PM defines a storage and privacy protection model by adding privacy regulations and considerations. The PM is designed to comply with the recent General Data Protection Regulation (GDPR) (Regulation (EU) 2016/679).

3.2.2 Implementation

Not applicable, the PM is owned by University of Trento and implementation details are not made public.

3.2.3 Interfaces, Endpoints and Resources Exposed

@Ronald: please check

We divide APIs in chapters. Let us start with the security ones:

- **POST /security/login** Used for authentication. The call includes as parameters an identifier and a password, and returns, among the other data, a token to be used for further interactions with the platform.
- **POST /security/logout** Log the peer out of the PM.
- **GET /person** Read the person of a given logged user.
- **GET /person_sl?username=:username&password=:password** Register a new human peer. Returns a peer ID.

The second set of APIs cover management operations:

- **GET /search**. General search machinery. Check [5] for more explanations on the query language to be used.
- **POST /collectives** Create collective: create a collective structure from a set of usernames.
- **GET /collectives/identifier** Return the usernames of all peers in a collective.
- **DELETE /collectives/** Delete the collective with a given id.

- **GET /types** Return the names and id of all the types defined in the peer manager.
- **GET /types/id** Return a list of all the attributes for the given type id.
- **POST /instances** Create a new instance from a given type and its attributes values.
- **GET /instances/id** Return the type and the attribute values for the given instance.
- **PUT /instances/id** Update one or more attributes for a given instance id (only attributes passed as in this call will be affected).
- **DELETE /instances/id** Delete the instance structure with the given id.
- **POST /profiles** Create a new profile from an existing instance, a transformation and a policy.
- **GET /profiles/id** Read profile by id: returns the type, the transformation, the policy and the attribute values for the given profile.
- **DELETE /profiles/id** Delete profile by id: delete the profile structure with the given id.
- **POST /transformations** Create a new transformation specifying the source type, the destination type and the attribute transformations to be performed.
- **GET /transformations/id** Read transformation by id: return the source type, the destination type, and the attribute transformations for the given transformation.
- **DELETE /transformations/id** Delete transformation by id: delete the transformation structure with the given id.
- **POST /policies** Create a new policy by specifying its use concept and validity limit.
- **GET /policies/id** Read policy by id: return the use concept and validity limit for the given policy.
- **DELETE /policies/id** Delete policy by id: delete the policy structure with the given id.
- **POST /peers** Create peer: create a new peer by specifying its main entity.

- **GET /peers/id** Read peer: read the main entity and defined profiles for a peer.
- **PUT /peers/id** Update peer profiles: update the defined profiles for a peer (replaces previous definitons).
- **DELETE /peers/id** Delete peer: delete the peer structure with the given id.
- **POST /users** Create user: create a new user by specifying the username(unique), password and main entity.
- **GET /users/id** Read user by id: read information stored in the user structure with the matching id.
- **GET /users/username** Read user by username: read information stored in the user structure with the matching username.
- **PUT /users/id** Update user password: replaces the existing user password for the user with the given id.
- **DELETE /users/id** Delete user: deletes the user with the given id.

3.2.4 Repository

Not applicable, the PM is owned by the University of Trento and not disclosed.

3.3 Context Manager

3.3.1 Functionality and Features

The Context Manager (CM) is the software/hardware embodiment of the three-layer approach for context recognition described in [5]. It includes three components:

- Sensing devices: able to capture relevant parameters for identifying the peer context.
- Modules for aggregating and merging attribute values from sensor data
- High level models stored in each peer and connected via dedicated APIs

The CM works as an extension of the PM and is used by the TEM to monitor parameters relevant to the task execution status and progress.

3.3.2 Implementation

Not applicable, the CM is owned by University of Trento and implementation details are not made public.

3.3.3 Interfaces, Endpoints and Resources Exposed

@Mattia/Ronald: please add

3.3.4 Repository

Not applicable, the CM is owned by the University of Trento and not disclosed.

3.4 Incentive Server

3.4.1 Functionality and Features

The Incentive Server analyzes in real-time user interaction and behaviour data within a single application and recommends incentives for these users. The IS includes learning methods, so that it can adapt its recommendations over time. The IS includes mechanisms for defining incentive types and messages, for receiving and storing behavioral data, for applying different algorithms on this behavioral data, for deciding on conditions that requires interventions and for recommending interventions for the applications that use it. See [**?**] for more details.

3.4.2 Implementation

The incentive server is written in Python, and uses the Django framework.

3.4.3 Interfaces, Endpoints and Resources Exposed

- **POST /login** Login Action. Used by external applications to obtain access to the intervention server (IS). Must be performed before any other operation access to the IS.
- **GET /api/incentive** Get all incentives types from the IS.
- **POST /api/incentive** Add a new incentive to the IS. Must specify scheme name/ID, type name/ID and conditions under which the incentive should be applied.

- **GET /getIncUser** Get the best incentive for a given user. This represents a pull operation by external application to obtain incentives for user.
- **GET /ask_by_date** Get intervention list from the given date.
  Up to now? Or just for the specific date?
- **/ask_by_id** Intervention or intervention list from the given id.
  Unclear: what is the ID referring to? User? Or what else?

The IS can be configured to work in push mode, sending incentives to an application; more details on how to do it are in [**?**]. In the same document the functioning of the endpoint for retrieving behavioural data and inputting it to the IS is also described.

3.4.4 Repository

The incentive server implementation can be found at: `https://gitlab.com/smartsociety/IncentiveServer`

3.5 Task Execution Manager

3.5.1 Functionality and Features

The TEM is tasked with coordinating the execution of tasks, especially those involving 'offline' actions by peers and collectives. The TEM acts as a monitoring platform and interacts with the Orchestration Manager and Context Manager/Peer Manager. It takes tasks agreed from the OM and instantiates the required monitors with the CM. For each task to be monitored the Execution Monitor requires as an input from the OM:

- A description of the task whose execution has to be monitored;
- The IDs of the peers involved in the execution

At the same time the TEM will also have access to:

- Sensory data related to the peers involved in the execution (from the CM) and;
- Mid and high-level data derived by sensor fusion in time and user input

Using these two sources the TEM produces an output to inform the OM of:

- The progress of a given tasks; and
- Possible deviations that occur during the execution of this task; major deviations should be properly expressed to allow the OM to react in a timely manner.

3.5.2 Implementation

The TEM is implemented in javascript using the node.js framework. Express is using for the REST API implementation and a MongoDB instance for storing data locally.

3.5.3 Interfaces, Endpoints and Resources Exposed

- **GET /monitorTask** Get all the tasks which are being monitored by the TEM.
- **POST /monitorTask** Starts monitoring the execution of an agreed task by instantiating the relevant monitors.
- **PUT /updateMonitor/:taskID** update the monitor of a task in TEM.
- **DELETE /terminateTaskMonitor/:taskID** Terminate the monitoring of a task.
- **GET /terminatedTasks** Get the list of all the terminated tasks.

3.5.4 Repository

`https://gitlab.com/smartsociety/taskexecutionmanager`

3.6 Provenance Service

3.6.1 Functionality and Features

Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness. The Provenance Service includes a specialised store for managing provenance records [4] and an aggregator of provenance types able to generate provenance summary reports. More details on the vocabulary to be used for HDA-CAS can be found in [**?**]. The provenance service allows to create and retrieve provenance templates and bindings, which are subsequently used for logging actions on the platform.

[4] ProvStore, `https://provenance.ecs.soton.ac.uk/store/`

3.6.2 Implementation

The provenance service is developed in Python and uses the Django framework.

3.6.3 Interfaces, Endpoints and Resources Exposed

- **POST template/** Post a template: Posts a provenance template to the Provenance Store (https://provenance.ecs.soton.ac.uk/store/). The data posted must contain key value pairs for prov and template_name.
- **GET template/:template/** Get a template: Returns a JSON object detailing the template's id, name, version and a URI linking to where the template is stored in the Provenance Store.
- **POST binding/** Post a provenance binding to the Provenance Store. The data posted must contain key value pairs for prov, template_name, and binding_name, it can optionally contain a version_id. When a binding is submitting without specifying a version, it is associated with the latest version of a template with template_name.
- **GET binding/:binding/** Get a binding: returns a JSON object detailing the binding's id, name, and a URI linking to where the binding is stored in the Provenance Store.
- **POST template/name/** Get all versions of a template: returns a JSON object containing all template versions with a specified name.

3.6.4 Repository

The `prov` package (a library for W3C provenance data model) is used by the PS and can be downloaded (MIT License) from `https://pypi.python.org/pypi/prov`. The PS codebase can be found at: `https://gitlab.com/smartsociety/submitbindings`

3.7 Reputation Service

3.7.1 Functionality and Features

The reputation service provides storage for and access to feedback and reputation reports, for SmartSociety applications. See [**?**] for additional details.

3.7.2 Implementation

The RS is implemented in Python and uses the Django framework and the prov library.

3.7.3 Interfaces, Endpoints and Resources Exposed

The API currently describes three resources, applications, feedback, and reputation, which are documented below.

- **GET application/:app/subject/byURI/:subject_uri/** Retrieve a subjects information by its URI. The subject uri must be percentage encoded. The response object includes a list of feedback reports about the subject and the current reputation report for a subject.
- **GET application/:app/subject/:subject/** Retrieve information about the subject with the given ID. The response object includes the subjects URI, a list of feedback reports about the subject and the the current reputation report for a subject.
- **POST application/:app/feedback/** Save a new feedback report.
- **GET application/:app/reputation/:reputation/** Retrieve the raw reputation report with the given ID.
- **application/:app/opinionOf/:author/aboutSubject/:subject/** Retrieve the raw reputation report about a subject authored by an author.

3.7.4 Repository

The reputation service implementation can be found at:`https://gitlab.com/smartsociety/reputationservice2.0/`

3.8 Communication Middleware

3.8.1 Functionality and Features

SmartCom provides communication functionality between a Hybrid Diversity-Aware Collective Adaptive System platform (HDA-CAS) on one side, and ICUs (Individual Computing Units, i.e., human-based services and software-based services) on the other side.

SmartCom provides low-level communication and control primitives that effectively virtualize the peers to HDA-CAS platform. It offers the asynchronous (message-based) communication functionality for interacting with dynamically-evolving collectives of peers both through native HDA-CAS applications, as well as through various third-party tools, such as Dropbox, Android devices, Twitter or email clients.

3.8.2 Implementation

The CM is implemented in java.

3.8.3 Interfaces, Endpoints and Resources Exposed

REST API

GET '/?type=¡type¿subtype=¡subtype¿' Get the message information for a message with a specific type and subtype POST '/' Add new message information for a message with a specific type and subtype GET /?type=¡type¿subtype=¡subtype¿

Returns the message information for a message with a specific type and subtype.

Parameter: type required - defines the type of the message subtype required - defines the subtype of the message

Respond: 200 message information instance is returned in the body 404 there is no message information for this type and subtype combination

POST /

Create a new message information for a given type and subtype (encoded in the body). If there is already such a message information, it will be overridden with this information.

Parameter: body required - instance of a message information JSON object

Respond: 200 resource has been created and the created message information is returned in the body

/** * Send a message to a collective or a single peer. The method assigns an ID to the message and * handles the sending asynchronously, i.e., it returns immediately and does not wait for the * sending to succeed or fail. Errors and exceptions thereafter will be sent to the Notification * Callback. Optionally, receipt acknowledgements are communicated back through the Notification * Callback API. * * The receiver of the message is defined within the message, it can be a peer or a collective. * * @param message Specifies the message that should be handled by the middleware. The receiver of the message is * defined by the message. * @return Returns the internal ID of the middleware to track

the message within the system. * @throws CommunicationException a generic exception that will be thrown if something went wrong * in the initial handling of the message. */ public Identifier send(Message message) throws CommunicationException;

/** * Add a special route to the routing rules (e.g., route input from peer A * always to peer B). Returns the ID of the routing rule (can be used to delete it). * The middleware will check if the rule is valID and throw an exception otherwise. * * @param rule Specifies the routing rule that should be added to the routing rules of the middleware. * @return Returns the middleware internal ID of the rule * @throws InvalidRuleException if the routing rule is not valid. */ public Identifier addRouting(RoutingRule rule) throws InvalidRuleException;

/** * Remove a previously defined routing rule identified by an Id. As soon as the method returns * the routing rule will not be applied any more. If there is no such rule with the given Id, * null will be returned. * * @param routeId The ID of the routing rule that should be removed. * @return The removed routing rule or null if there is no such rule in the system. */ public RoutingRule removeRouting(Identifier routeId);

/** * Creates a input adapter that will wait for push notifications or will pull for updates in a * certain time interval. Returns the ID of the adapter. * * @param adapter Specifies the input push adapter. * @return Returns the middleware internal ID of the adapter. */ public Identifier addPushAdapter(InputPushAdapter adapter);

/** * Creates a input adapter that will pull for updates in a certain time interval. * Returns the ID of the adapter. The pull requests will be issued in the specified * interval until the adapter is explicitly removed from the system. * * @param adapter Specifies the input pull adapter * @param interval Interval in milliseconds that specifies when to issue pull requests. Cant be zero or negative. * @return Returns the middleware internal ID of the adapter. */ public Identifier addPullAdapter(InputPullAdapter adapter, long interval);

/** * Creates a input adapter that will pull for updates in a certain time interval. * Returns the ID of the adapter. The pull requests will be issued in the specified * interval. If deleteIfSuccessful is set to true, the adapter will be removed in case of * a successful execution, it will continue in case of a unsuccessful execution. * * @param adapter Specifies the input pull adapter * @param interval Interval in milliseconds that specifies when to issue pull requests. Cant be zero or negative. * @param deleteIfSuccessful delete this adapter after a successful execution * @return Returns the middleware internal ID of the adapter. */ public Identifier addPullAdapter(InputPullAdapter adapter, long interval,

boolean deleteIfSuccessful);

/** * Removes a input adapter from the execution. As soon as this method returns, the * adapter with the given ID will not be executed any more. It returns the requested * input adapter or null if there is no adapter with such an ID in the system. * * @param adapterId The ID of the adapter that should be removed. * @return Returns the input adapter that has been removed or nothing if there is no such adapter. */ public InputAdapter removeInputAdapter(Identifier adapterId);

/** * Registers a new type of output adapter that can be used by the middleware to get in contact with a peer. * The output adapters will be instantiated by the middleware on demand. Note that these adapters are required * to have an @Adapter annotation otherwise an exception will be thrown. * In case of a stateless adapter, it is possible that the adapter will be instantiated immediately. If * any error occurs during the instantiation, an exception will be thrown * * @param adapter The output adapter that can be used to contact peers. * @return Returns the middleware internal ID of the created adapter. * @throws CommunicationException if the adapter could not be handled, the specific reason is embedded in * the exception. * @see at.ac.tuwien.dsg.smartcom.adapter.annotations.Adapter */ public Identifier registerOutputAdapter(Class¡? extends OutputAdapter¿ adapter) throws CommunicationException;

/** * Removes a type of output adapters. Adapters that are currently in use will be removed * as soon as possible (i.e., current communication wont be aborted and waiting messages * in the adapter queue will be transmitted). * * @param adapterId Specifies the adapter that should be removed. */ public void removeOutputAdapter(Identifier adapterId);

/** * Register a notification callback that will be called if there are new input messages * available. * * @param callback callback for notification * @return returns the identifier of the callback (can be used to remove it) */ public Identifier registerNotificationCallback(NotificationCallback callback);

/** * Unregister a previously registered notification callback. * * @param callback callback for notification */ public boolean unregisterNotificationCallback(Identifier callback);

3.8.4 Repository

`https://gitlab.com/smartsociety/SmartCom/`

3.9 Monitoring

3.9.1 Functionality and Features

The implementation of the monitoring component is based upon three basic components: Modules gathering and publishing the relevant information; Modules consuming the monitoring related information; Information dissemination infrastructure. In particular, local agents collect specific information regarding the liveliness and health status of each component and communicate with the infrastructure via lightweight java/js clients. The monitoring infrastructure provides scalable support to the collection of local agents feeds. A monitoring dashboard represents the main consumer of monitoring data generated by the agents and dispatched through the monitoring infrastructure. The implementation shipped with the i-locate toolkit, whose architecture is sketched in Figure 4, is based on the logstash open source framework , which presents very good support for collection of logs in various schemas/formats, and has a large number of plugins available for the most widely used commercial frameworks. The usage of logstash is coupled with a redis broker for additional scalability and elasticsearch for indexing and persistence. The usage of redis broker is optional, currently not foreseen for the limited-scale i-locate pilots. Aggregated and curated logs are stored in the elastic database and can be queried via standard interfaces, enabling technical supporting partners to develop their own ad hoc monitoring dashboard or to integrate with legacy ones. The default option for the i-locate toolkit is to use Kibana, a flexible dashboard which supports seamless integration with elasticsearch and presents basic, yet sufficient analytics functionality. A sample dashboard is reported in Figure 5.

The goal of the monitoring component is to enabler system administrators to monitor the liveliness of the SmartSociety platform components, possibly distributed across multiple servers. Target users of the functionality exposed are therefore system administrators at pilot sites and i-locate technical supporting partners. The main expected usage is for troubleshooting in case of platform malfunctioning, alerts and warnings. In the long term, it can enable the deployment of self-healing mechanisms.

3.9.2 Implementation

The monitoring framework is based on Logstash[5], an open-source tool for managing events and logs. Logstash can be used to collect logs, parse them, and store them for later use

[5] `http://logstash.net/`

(e.g., search and visualization). It is possible to define what information shall be permanently stored and processed by the monitoring framework. In particular, it is possible to integrate:

- System logs: these logs correspond to logs, which are generated by the various system components such as, e.g., web servers, application servers.
- Application logs: specific logs that are produced by applications, and require a constant integration for debugging and monitoring purposes.
- Monitoring information: any agent that can be configured to deliver data to the Logstash infrastructure.

In all three cases, a Logstash shipper is used to connect the specific source of data to Logstash. Specific shippers already exists for some widely used system components such as, e.g., web servers, databases, etc., while custom shippers can be created for specific cases. In the case of SmartSociety, we created a dedicated shipper to collect the events produced by the various platform components.
The following component is a Redis Broker. This is an optional component that can be used in order to scale the system to large volumes of events and data. Based on Redis, data is indexed in order to prepare it for optimal searching and querying. Once the data is indexed, it is stored in an ElasticSearch cluster for storage and search. Starting from the data stored in ElasticSearch, it is possible to build queries on scale to explore the collected data. We decided to use Kibana[6] as the tool to create and visualize queries on the collected data. Kibana is fully integrated with ElasticSearch, and allows to easily explore and analyze large volumes of data. A sample visualization is shown in **??**. The metrics and specific charts can be configured dynamically by the administrator of the platform.

3.9.3 Interfaces, Endpoints and Resources Exposed

The Monitoring component functionality is accessible through the Kibana GUI. Elasticsearch Search APIs can be used for integration with legacy visualization dashboards. Otherwise, a Kibana dashboard is shipped.

[6] `https://www.elastic.co/products/kibana`

3.9.4 Repository

Not applicable.

add to the repo empty project with wiki with instructions on how to build it

A Smart Society Application has its own identifier, generated during the registration phase. Currently, the application code has to be written by a developer directly in Java; in the final version of the platform, the code will be generated dynamically through the programming framework.

3.10 Runtime

Each application is expected to run in its own process by the SmartSociety runtime. The application will be provided by runtime with a SmartCom and a Orchestrator instances. In order to interact with external entities (such as peers or user applications), the runtime exposes three kinds of resources:

- **POST:/task/:applicationId/** To submit tasks, the runtime will ask the application to create an *instance initializer* that will take care of the setup phase of the task. The application is expected to carry out all the operation that can be performed before interacting with the peers. The posted data is domain–dependent, and it will be serialized and managed by the application at the moment of instance creation. The runtime associated to the instance (hence the task) creates an identifier that is returned as response for further usage.
- **GET:/task/:applicationId/** This resource is used to query the status of one or multiple tasks. Query parameters can be used to specify a filter on the tasks of interest. The status format for each task is just required to be a valid json node (either a text or more complex structures) and it is completely domain–dependent.
- **GET:/task/:applicationId/:taskId** Used to retrieve the status of a specific task. As for the previous point both query parameters and the response are domain–dependent.
- **POST:/message/:applicationId/** Used by the peer applications to communicate back with the application. To use this endpoint the application must have previously contacted the peer with a message containing a given conversation identifier. Such identifier, which is passed as a specific field in the message body, is used by the

runtime to dispatch the message to the correct application instance. The content of the message is then handled by the task runner instance that changes its state according to the information received.

In v.2.0 of the platform, the application developer is expected to provide certain functionalities. This includes methods for:

- Creating a new task runner that will take care of setting up and carry out the task submitted to the application;
- Retrieve the active task runners according to some query parameters, this will be used by the query endpoint.

The task runner is the instance of an application-specific class, which implements a simple interface allowing the runtime to start the execution and to ask for its status.

3.11 Component Library

The component library allows the application development to be abstracted from the actual implementation of the components. Component wrappers provide a simple interface to the component integrated with the runtime, so that the developer is shielded from minor changes in components.

In this section we briefly describe the functionality provided by the component library.

3.11.1 SmartCom

The developer does not need to interact directly with SmartCOM: a method is provided for sending a message to a given collective, the programmer is required to provide a specific handler for handling answers to the message. The runtime will take care of receiving answers through the specific REST endpoint and, by using the conversation ID, it will route the answer to the correct handler. The platform provides also an adapter for sending out Android notifications or for contacting peers trough a REST endpoint. Other supported adapters (Dropbox, Email, ...) can be added easily.

3.11.2 Orchestration Manager

When the application is launched an orchestration manager (OM) is also executed. Communication with the OM works trough a REST API (a generalized version of the one presented in [4]). The library supports the following requests:

- Submit a task request for composition;
- Retrieve a specific task request or task;
- Wait for a negotiable task;
- Accept a task on behalf of a peer;
- Perform a trivial negotiation with explicit agreement;
- Wait for an agreed task.

3.11.3 Peer Manager

The library allows the easy creation of a collective given a collection of peers. The identifier of the collective is the only information needed by SmartCom to carry out communications with all the relevant peers.

3.11.4 Provenance

The runtime provides easy methods for logging on the provenance store the generic (i..e, independent from the application domain) part of the provenance graph. This includes actions such as, e.g., creation of a task, creation of a collective etc. The domain-dependent part of the provenance graph shall be specified by the application developer.Some helper function is provided for binding specific data whose model cannot be known a priori. The communication with provenance store (happening through a REST API) is hidden to the developer by the library.

3.12 Monitoring Framework

The monitoring framework is responsible for the monitoring of the overall SmartSociety platform and components. It acts as a central collection point for any information which is considered important to ensure the proper functioning of the platform. More details are reported in App. **??**.

3.13 Expected evolution

In the future version of the platform each application and its runtime will run in separate containers. This will provide isolation among different applications and will support faster

application deployment and portability across machines. A service for routing the requests from general endpoints to the right application runtime will be included.

The way the runtime will evolve and its interaction with applications are highly dependent on the outcome of the programming model and programming framework specification efforts currently ongoing within WP7.

In terms of integration of other components, the roadmap foresees to integrate in the upcoming months the context manager (which will interact strictly with the peer manager), the reputation maanager, the task execution manager and the incentive server.

4 Expected Usage and Examples

A Smart Society Application has its own identifier, generated during the registration phase. Currently, the application code has to be written by a developer directly in Java; in the final version of the platform, the code will be generated dynamically through the programming framework.

take from D6.2

Programmers are expected to interact with the SmartSociety platform functionality through the programming framework (PF). While direct usage of the component-level APIs described in the previous section is possible, it is deprecated. The PF instantiates the programming model defined in [9], that we briefly summarise here for the sake of consistency.

The key notion is that of a Collective-Based Task (CBT), an object encapsulating all the necessary logic for managing complex collective-related operations: team provisioning and assembly, execu- tion plan composition, human participation negotiations, and finally the execution itself. These operations are provided by various SmartSociety platform components, which ex- pose a set of APIs used by the programming model libraries. During the lifetime of a CBT, various Collectives related to the CBT are created and exposed to the developer for further (arbitrary) use in the remainder of the code, even outside of the context of the originating CBT or its lifespan.

At CBTs core is a state machine (Fig. 3) managing transitions between states representing the eponymous phases of the task lifecycle: provisioning, composition, ne- gotiation and execution. An additional state, named continuous orchestration, is used to represent a state combining composition and negotiation under specific conditions, as explained in Table 1.

The notion of collective is very general. Sometimes it used to denote a stable group or category or peers based on the common properties, but not necessarily with any personal/professional relationships (e.g., Java developers, students, Milano residents, peers registered with a specific application); in other cases, the term collective was used to refer to a team a group of people gathered around a concrete task. The former type of collectives is more durable, whereas the latter one is short-lived. Therefore, we make following distinction in the programming model: A Resident Collective (RC) is an entity defined by a persistent PeerManager iden- tifier, existing across multiple application executions, and possibly different applications. Resident collectives can be created, altered and destroyed

also fully out of scope of the code managed by the programming model. The control of who can access and read a resident collective is enforced by the PeerManager. For those resident collectives accessible from the given application, a developer can read/access individual collective members as well as all accessible attributes defined in the collectives profile. When accessing or creating a RC, the programming model either passes to the PeerManager a query and constructs it from returned peers, or an ID to get an existing PeerManager collective. In either case, in the background, the programming model will pass to the PeerManager its credentials. So, it is up to the PeerManager to decide based on the privacy rules which peers will get exposed (returned). For example, for TrentoResidents we may get all Trento residents who have nothing against participating in a new (our) application, but not necessarily all Trento residents from the PeerManagers database. By default, the newly-created RC remains visible to future runs of the application that created it, but not to other applica- tions. PeerManager can make them visible to other applications as well. At least one RC must exist in the application, namely the collective representing all peers visible to the application. Application-Based Collective (ABC). Differently from a resident collective, an ABCs lifecycle is managed exclusively by the SmartSociety application. Therefore, an ABC cannot be accessed outside of an application, and ceases to exist when the appli- cations execution stops. The ABCs are initially created by applying certain operations on resident collectives. Differently than resident collectives, an ABC is an atomic entity for the developer, meaning that individual peers cannot be explicitly known or accessed from an ABC instance. This means that an ABC instance is immutable. The ABC is the 20 of 48 http://www.smart-society-project.eu

Deliverable 7.2 c SmartSociety Consortium 2013-2017 embodiment of the principle of collectiveness, making the collective an atomic, first-class citizen in our programming model. Concretely, the following rules regulate the ABCs lifecycle: An ABC can be created: implicitly, as intermediate products of different states of CBT execution (e.g., provisioned, agreed). explicitly, by using dedicated collective manipulation operators: to clone a resident collective. as a result of a set operation on two other Collectives (either RC or ABC). An ABC has a lifetime equal to that of the application where it was created. the lifecycle of an ABC is managed by the SmartSocietyApplicationContext (CTX) (Sec. 2.4) Developer cannot explicitly delete them; they are cleaned up by the runtime when the object representing the application is finalized. An ABC is: immutable for the developer. atomic for the developer (i.e., individual peers cannot be accessed from it). The context object (Sec. 2.4) registers and keeps the collectives kind.

The application context (CTX) represents a particular instance of the SmartSociety ap- plication (class SmartSocietyApplicationContext). It can be described in more detail with the following claims: There must be exactly 1 CTX instance per application run. Therefore, CTX is singleton. CTX instance keeps all values, overall application state and metadata initialized for the application run. Therefore, all initialization methods are over CTX. CTX represents the application, therefore it possesses the internal business logic to track lifecycles of all created ABC instances, and other entities created for the purpose of the application.

During the CTX initialization the developer provides the required information that will be used during the application execution (Section 2.4). The programming model allows the developer to submit such information through the following methods: void updateInternaProfile(ProfileSchema profileSchema, Profile profile): the internal profile is created and updated, in case of creation the profileSchema is also provided to the peer-manager; void registerUserProfileSchema(ProfileSchema profileSchema): the schema of the profile that will be used to store user data; void collectiveForKind(String kind, CollectiveDescriptor descriptor): for each collective kind a corresponding descriptor is registered; void registerBuilderForCBTType(String cbtType, CBTBuilder builder): this method specify how CBT of a certain type can be built. Several CBTBuilder implementations can be provided as part of the programming model library, in par- ticular there is one based on OMProxy to make use of an external Orchestration Manager.

The CBT is instantiated through a CBTBuilder. Builders are registered at CTX initialization time and associated to a CBT type (Sec- tion 3.1). Once the right type builder is retrieved all the expected parameters must be passed to the builder (according to its actual type), and then the build must be called (as shown in Listing 1). CBT cbt = ctx.getCBTBuilder("RideSharingType") 3 .of(CollaborationType.OC) //Enum: OC, OD, $OC_OD4.forInputCollective(c)5.forTaskRequest(t)6.withNegotiationArgs(myNegotiationArgs)7.build$

he basic method for checking the lifecycle state is the following. It returns an enumeration CBTState with the active state of the CBT. CBTState getCurrentState() Returns the currents state. We additionally provide the following utility (pretty-print) methods for comparing the state of execution: boolean isAfter(CBTState compareWith) returns true if the CBT has finished executing the compareWith state; this also includes waiting on the subsequent state. Throws exception if the comparison is illogical. boolean isBefore(CBTState compareWith) returns true if the CBT has not yet started executing the compareWith state; this also includes waiting for the com- pareWith state. Throws

exception if the comparison is illogical. boolean isIn(CBTState compareWith) checks if compareWith is equal to the return value of getCurrentState().

Upon instantiating a CBT, the developer defines whether the state transitioning should happen automatically, or be explicitly controlled. In order to check for these states, we expose the following set of methods: boolean isWaitingForProvisioning() boolean isWaitingForComposition() boolean isWaitingForNegotiation() boolean isWaitingForContinuousOrchestration() boolean isWaitingForStart() waiting in the initial state to enter any main state. Furthermore, we have the following related methods: boolean isRunning() true in every other state except initial or final. boolean isDone() true only in final state (either success or fail), not matter whether we arrived in it through success or one of the fail states.

To allow the developer to control CBT transitions explicitly the developer is offered the following constructs to get/set the flags used in guarding conditions and wake up the CBTs thread if it was waiting on this flag: get/setDoCompose(boolean tf) get/setDoNegotiate(boolean tf) get/setDoExecute(boolean tf) By default, the CBT gets instantiated with all flags set to true. We also provide a convenience method that will set at once all flags to true/false: setAllTransitionsTo(boolean tf)

Since from the initial state we can transition into more than one state, for that we use the method: void start() allows entering into provisioning or continuous orchestration state (depending which of them is the first state). Non-blocking call. Additionally, CBT exposes a number of additional methods to match up to the methods offered by the Java 7 Future API: TaskResult get() waits if necessary for the computation to complete (until is- Done() == true), and then retrieves its result. Blocking call. TaskResult get(long timeout, TimeUnit unit) same as above, but throwing appropriate exception if timeout expired before the result was obtained. boolean cancel(boolean mayInterruptIfRunning) attempts to abort the over- all execution in any state and transition directly to the final fail-state. The original Java 7 semantics of the method is preserved. boolean isCancelled() Returns true if CBT was canceled before it completed. The original Java 7 semantics of the method is preserved.

A CBT object exposes the following methods for fetching the ABCs created during CBTs lifecycle: Collective getCollectiveInput() returns the collective that was used as the input for the CBT. ABC getCollectiveProvisioned() returns the provisioned collective ABC getCollectiveAgreed() returns the agreed collective. List¡ABC¿ getNegotiables() returns the list of negotiable collectives.

As noted in Section 2.3 resident collectives (RCs) are created by querying the PeerManager via the following static methods of the ResidentCollective class: ResidentCollective createFromQuery(PeerMgrQuery q, string to kind) Cre- ates and registers a collective with the PeerManager. It is assumed that the kind entity describing the new collective has been properly registered and initialized with CTX. When contacting PeerManager we pass also the ID of the application, and we assume that PeerManager checks (with the help of the programming model run- time) whether we are allowed to create a collective of the requested kind, and filters out only those peers whose privacy settings allow them to be visible to our appli- cations queries. The registered kind descriptor in the CTX allows this method to know how to properly transform the attributes from the entities obtained from the PeerManager to those expected by the target kind. ResidentCollective createFromID(string ID, string to kind) Creates a lo- cal representation of an already existing collective on the PeerManager, with a pre- existing ID. Invocation of this method will not create a collective on PeerManager, so in case of passing a non-existing collective ID an exception is thrown. This method allows us to use and access externally defined RCs. When contacting PeerManager we pass also the ID of the application, and we assume that the PeerManager checks 32 of 48 http://www.smart-society-project.eu

Deliverable 7.2 c SmartSociety Consortium 2013-2017 whether we are allowed to access the requested a collective, and filters out only those peers whose privacy settings allow them to be visible to our applications queries. The registered kind descriptor in the CTX allows this method to know how to prop- erly transform the attributes from the entities obtained from the PeerManager to those expected by the target kind. On the other hand, ABCs are created from existing collectives (both RCs and ABCs) through the following static methods of the Collective class: ABC copy(Collective from, [string to kind)] Creates an ABC instance of kind to kind. Peers from collective from are copied to the returned ABC instance. If to kind is omitted, the kind from collective from is assumed. ABC join(Collective master, Collective slave, [string to kind)]) Cre- ates an ABC instance, containing the union of peers from Collectives master and slave. The resulting collective must be transformable into to kind. The last argu- ment can be omitted if both master and slave have the same kind. ABC complement(Collective master, Collective slave, [string to kind)]) Creates an ABC instance, containing the peers from Collective master after re- moving the peers present both in master and in slave. The resulting collective must be transformable into to kind. The last argument can be omitted if both master and slave have the same kind. void persist() Persist the collective on PeerManager. RCs are

already persisted, so in this case the operation defaults to renaming. In case of an ABC, the PeerMan- ager persists the collective as RC. However, this does not mean that the developer is able to subsequently fetch that RC and access the collective members. This decided by the CTX and PeerManager based on the ABCs kind. For reading the ABCs attributes, the following ABC class method is used: Attribute GetAttribute() searches attributes fields then returns a clone of the found Attribute, if any present. c SmartSociety Consortium 2013-2017 33 of 48

c SmartSociety Consortium 2013-2017 Deliverable 7.2 3.6 Collective-level communication Programming model fully relies on the messaging and virtualization middleware Smart- Com9 developed within T7.1 for supporting the communication with peers and collectives. Programming model allows at the moment only a basic set of communication constructs, namely those for sending a message to a hybrid collective, and receiving responses from it: void send(Message m) throws CommunicationException Send a message to the collective. Non-blocking. Does not wait for the sending to succeed or fail. Errors and exceptions thereafter will be sent to the Notification Callback. Identifier registerNotificationCallback(NotificationCallback onReceive) Register a notification callback method that will be called when new messages from the collective are received. The returned Identifier is used for unsubscribing. void unregisterNotificationCallback(Identifier callback)unregisterapre- viously registered callback. These methods are invokable on any Collective object.

3.7 Examples 3.7.1 Initializing a CBT void ctx$_i nitialization()/*RegisteringOMProxybasedbuilderforC$
$thismethodiscalledforrequestsrequiringa"RideSharing"CBT*/CBTget_CBT_for_rideSharing(TaskRequest$
$thismethodiscalledforrequestsinvolvinga"SimpleTask"request*/CBTget_CBT_for_rideSharing(TaskReque$
$CBTinitializationListing2showsanexampleofhowaCBTcanbeinitialized.Thepiecesofcodearegroupedindu$
$11)isexecutedaftertheCTXhasbeencre-ated.Thedevelopermustprovideа CBTBuilderforeachtypeofCBTt$
$ity,asupportlibraryprovidedalongwiththeprogrammingmodelwillprovideseveralCBTBuilderimplementat$
$201735of48cSmartSocietyConsortium2013-2017Deliverable7.2ForthetypeRideSharingthedeveloperdecid$
$21fortheRideSharingtype,lines23-32forSimpleTask),onecanappreciatetheconsistencyoftheCBTBuilde$
$hoc,on-demandprogrammingteamstobuildsoftwareartifacts.Forthispurpose,twoCBTtypesareregistered$
$MyJavaProgrammingTaskandMyJavaTestingTask.First,thedevelopercreatesaRCjavaDevscontaining$
$Manager.Thiscollectiveisusedastheinputcollectiveoftheprog TaskCBT.progTaskisinstantiatedasanon-$
$demandcollectivetask,meaningthatthecompositionstatewillbeskipped,sincetheexecutionplaninimpliedfro$
$merswithparticularskillsareselectedandajointcoderepositoryissetforthemtouse.Theoutputoftheprovision$
$builtABCcollective,containingtheselectedprogrammers.Sinceitisatomicandimmutable,theexactprogramm$

$//www.smart-society-project.eu$

Deliverable 7.2 c SmartSociety Consortium 2013-2017 lected into the agreed one. This collective is then used to perform the second CBT testTask, which takes as input the output of the first CBT. 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 ... assume negotiation on progTask done ... Collective testTeam; //will be ABC if (progTask.isAfter(CBTState.NEGOTIATION)) // out of provisioned developers, use the other half for testing testTeam = Collective.complement(progTask.getCollectiveProvisioned(), progTask.getCollectiveAgreed()); while (!progTask.isDone()) /* do stuff or block on condition */ TaskResult progTRes = progTask.get(); if (! progTRes.isQoRGoodEnough()) return; CBT testTask = ctx.getCBTBuilder("MyJavaTestingTask") .of(CollaborationType.OD)
1 2 3 4 5 6 7 8 9 Collective javaDevs = ResidentCollective.createFromQuery(myQry,"JAVA$_{D}EVS$"); $CBTpr$
$ctx.getCBTBuilder("MyJavaProgrammingTask").of(CollaborationType.OD).forInputCollective(jav$
$...*/.forInputCollective(javaDevs).forTaskRequest(newTaskRequest(progTRes)).withNegotiationAr$
$Usingtheagreedandprovisioned ABCstoobtainathirdcollectivethatwillbeusedinanothertask.Also, usingth$

The following code snippet shows some examples of interacting with CBT lifecycle. An on-demand CBT named cbt is initially instantiated. For illustration purposes we make sure that all transition flags are enables (true by default), then manually set do negotiate to false, to force cbt to block before entering the negotiation state, and start the CBT. While CBT is executing, arbitrary business logic can be performed in parallel. At some point, the CBT is ready to start negotiations. At that moment, for the sake of demon- stration, we dispatch the motivating messages (or possibly other incentive mechanisms) to the human members of the collective, and let the negotiation process begin. Finally, we block the main thread of the application waiting on the cbt to finish or the specified timeout to elapse, in which case we explicitly cancel the execution. 1 2 3 4 5 6 7 8 9 CBT cbt = /*... assume on$_{d}emand = true... *$
$/cbt.setAllTransitionsTo(true); //optionalcbt.setDoNegotiate(false); cbt.start(); while(cbt.isRunning()$
$cbt.getNegotiables()negotiatingCol.send(newSmartCom.Message("Pleaseacceptthistask")); //negotiat$

5 Validation

The final goal of validation activity is to test the ability of the platform to meet the high-level requirements described and analyzed in detail in [1]. In Table 1 we summarize the current status of the platform development in terms of ability to meet the requirements identified by the Consortium. As it can be seen, not all requirements have been met yet,

in most cases as they refer to components which are still not fully completed within the respective workpackage. Yet, the platform in its current version already provides the minimum level of functionality required to handle a rather large range of social computations, as demonstrated by the available demo applications.

We now move to briefly describing the validation and testing activities carried out within the scope of Task T8.4 ('Lab experiments and platform validation'). These were based on the flexible AskSmartSociety! application developed in year-2 and presented in [2] and included the development of a number of small-scale prototypes aimed at testing the flexibility and extensibility of the platform. In particular, the following two results were achieved:

- **Image tagging application:** based on the AskSmartSociety! application, a simple image tagging application was carried out. By means of such an application participants can post an image through the Ask! mobile application and ask other participants to provide tags annotating the image content. Other participants can tag the image using the Reply! application or the SmartSociety twitter peer. Tags are collected in the AskSmartSociety! peer, where they are disambiguated (using a third-party semantic analysis engine), ranked and finally the most likely tags are reported back to the user. The development required minimal modifications of the Ask! and Reply! application, integration with a third-party service for entity extraction [7] and a third-party service for image storage [8].
- **Facebook Peer and AskSmartSociety! Integration:** we developed a Facebook peer, in the form of a Facebook application connected to the SmartSociety platform. The integration was carried out for the AskSmartSociety! application. In this case, as it was for the Twitter peer, questions asked through the Ask! application get posted on the Facebook app page. Users who subscribed to the application can reply to the question using the 'comment' Facebook functionality. Additionally, users can 'like' comments posted by other users, an information which may be used to rank answers. In the AskSmartSociety! peer, answers from the Facebook peer are collected and integrated with those coming from the Reply! mobile app and the Twitter live feed.

The work carried out to realize the two prototypes highlighted a number of issues in the

[7] `https://dandelion.eu/`
[8] `http://imgur.com/`

ID	Chapter	Description	Met?	Remarks
CR-1	Computational Requirements	The platform should be able to execute human-based computations	Yes	See SmartShare and AskSmartSociety! demo
CR-2	Computational Requirements	The platform should be able to execute machine-based computations	Yes	
CR-3	Computational Requirements	The platform should be able to support hybrid computations	Yes	See AskSmartSociety! demo
PP-1	Peer Profiles and Peer Profiling	Peers will be characterized by a static and a dynamic profile.	Partially	See [10, 11, 5], integration with context manager ongoing
PP-2	Peer Profiles and Peer Profiling	Peer profiles data storage	Partially	See [5] for integration with PPL
PP-3	Peer Profiles and Peer Profiling	Peers can have multiple profiles	Partially	See [11, 5]
PP-4	Peer Profiles and Peer Profiling	Platform will support the profiling of peers	Partially	Integration with reputation service completed, full profiling missing
PU-1	Platform Usability	The SmartSociety platform should be accessible through a set of open APIs	Partially	Final set of APIs will depend on the programming framework [9]
PU-2	Platform Usability	Ease for developers to create and manage applications	No	Depends on release of programming framework
PU-3	Platform Usability	Support application development and deployment life-cycle automatically	Partially	A first version of the runtime is included in v2.0
PU-4	Platform Usability	Management GUI	Yes	The monitoring component has been integrated in v2.0 and includes a management dashboard
HT-1	Platform Components and Interactions Heterogeneity	Number of different channels to interact with human peers	Yes	See [12] and AskSmartSociety! demo.
HT-2	Platform Components and Interactions Heterogeneity	Platform components will run on a variety of heterogeneous devices	No	Currently tested only on servers.
HT-3	Platform Components and Interactions Heterogeneity	SmartSociety applications will be accessible via a variety of devices, online via the web and on mobile devices	Yes	See SmartSmare and AskSmartSociety! demos.
HT-4	Platform Components and Interactions Heterogeneity	SmartSociety will support a diversity of user interfaces for user engagement and recruitment	Partially	See [13]
SEC-1	Privacy and Security	Access Control	Partially	See [5]
SEC-2	Privacy and Security	Trust and Reputation	Partially	Based on integration of the reputation service
SEC-3	Privacy and Security	Informed Consent	Partially	See [5]
SEC-4	Privacy and Security	Peer Defined Usage Control Policies	No	
SEC-5	Privacy and Security	Secure Collection and Storage	Yes	See [5]
SEC-6	Privacy and Security	Comprehensive explanations of security and privacy issues	No	
GOV-1	Governance	Platform should support the collection of data related to governance	Partially	Provenance service integrated in v.2.0
GOV-2	Governance	Platform should support the implementation of governance policies	No	
PR-1	Performance Requirements	Scalability	No	No scalability tests performed so far.

Table 1: Features of platform 2.0 against requirements.

current version of the platform, in particular in terms of functionality and ease of use of the APIs exposed by the Peer Manager; a new set of APIs was released by WP4 accordingly.

6 Conclusions

The vision of the SmartSociety platform is to become an 'IFTTT [9] for social computation', i.e., a platform that allows the easy integration of hybrid (human and machine) computational elements within the scope of a well-defined application workflow. While various steps have been taken in this direction, the overarching goal will require intense activities for the remainder of the project.

Version 2.0 of the SmartSociety platform integrates seven key components: peer manager, orchestration manager, communication middleware, provenance service, reputation service, monitoring and analysis service and application runtime. This configuration is sufficient for developing a number of different applications using various types of social computation. Some limitations in terms of the single components are expected to be overcome with refinements from the relevant technical WP.

In terms of the upcoming steps, four major components to be integrated are the context manager (which will interact strictly with the peer manager), the reputation maanager, the task execution manager and the incentive server. Major changes in the platform configuration will come from the delivery of the programming framework by WP7 (foreseen at M36), which will have a major impact on how platform functionality can be accessed by application developers. Interactions with the virtual gamified environment in WP9 may also lead to a refactoring of some of the features exposed by the platform. Another major release of the platform is foreseen at M36.

Taken from D1.2

There is here an interesting element related to the deployment (and business) model of the platform. If the platform is made available 'as a service', indeed, the platform controller can exercise some form of control, for example by means of a code of conduct (formalised through, e.g., terms and conditions for the usage of the platform services). This could be use to perform some control on what the platform is used for, and eventually ban some applications which are deemed not in line with the ethical principles of the SmartSociety project. On the other hand by providing the platform as an open-source server that anybody can setup and deploy there are no means to enforce any form of control, so that the platform could, at least theoretically, be used for managing and operating 'evil' (read it: discriminatory, non democratic etc.) HDA- CASs. There is therefore a tension here,

[9] `https://ifttt.com/`: IFTTT (IF This Then That) is a Web-based service allowing non-technical users to create chains of conditional statements (called 'IF recipes') and actions (called 'DO recipes') involving popular Web services.

in that both models bring about advantages and disadvantages, which should be properly balanced.

References

[1] I. Carreras, "Platform requirements analysis and system design," SmartSociety FP7 Project Deliverable D8.1, Dec. 2013.

[2] I. Carreras, D. Miorandi, and T. Schiavinotto, "Platform prototype: early results and progress report," SmartSociety FP7 Project Deliverable D8.2, Dec. 2014.

[3] O. Scekic, D. Miorandi, T. Schiavinotto, D. I. Diochnos, A. Hume, R. Chenu-Abente, H.-L. Truong, M. Rovatsos, I. Carreras, S. Dustdar, and F. Giunchiglia, "SmartSociety – a platform for collaborative people-machine computation," in *Proc. of SOCA 2015*, Rome, IT, 2015.

[4] M. Rovatsos, "Static social orchestration: implementation and evaluation," SmartSociety FP7 Project Deliverable D6.2, Dec. 2014.

[5] A. H. Daniele Miorandi and R. Chenu, "Peer modeling and search," SmartSociety FP7 Project Deliverable D4.3, Jun. 2015.

[6] G. Kampis, "Models for human/machine symbiosis: final results," SmartSociety FP7 Project Deliverable D3.2, Oct. 2014.

[7] M. Rovatsos, "Computational models and validation," SmartSociety FP7 Project Deliverable D2.3, Jun. 2015.

[8] L. Moreau, "Provenance, trust, reputation and big data," SmartSociety FP7 Project Deliverable D2.2, Dec. 2014.

[9] O. Scekic and H.-L. Truong, "Programming models and languages," SmartSociety FP7 Project Deliverable D7.2, Jun. 2015.

[10] A. Hume, "Peers and micro-task profiling: early results and progress report," SmartSociety FP7 Project Deliverable D4.1, Dec. 2013.

[11] ——, "Peer search in smart societies," SmartSociety FP7 Project Deliverable D4.2, Dec. 2014.

[12] H.-L. Truong and O. Scekic, "Virtualization techniques and prototypes," SmartSociety FP7 Project Deliverable D7.1, Dec. 2014.

[13] M. Pompa, "Prototype version 1, testing scenarios and initial evaluation," SmartSociety FP7 Project Deliverable D9.3, Jun. 2015.