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 10, 2016*
Status[2]	*D*
Total Number of pages:	38
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.
CBT	Collective-Based Task	Object encapsulating all the necessary logic for managing complex collective-related operations
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 [1]. 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 [2]. 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 [4]:

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.

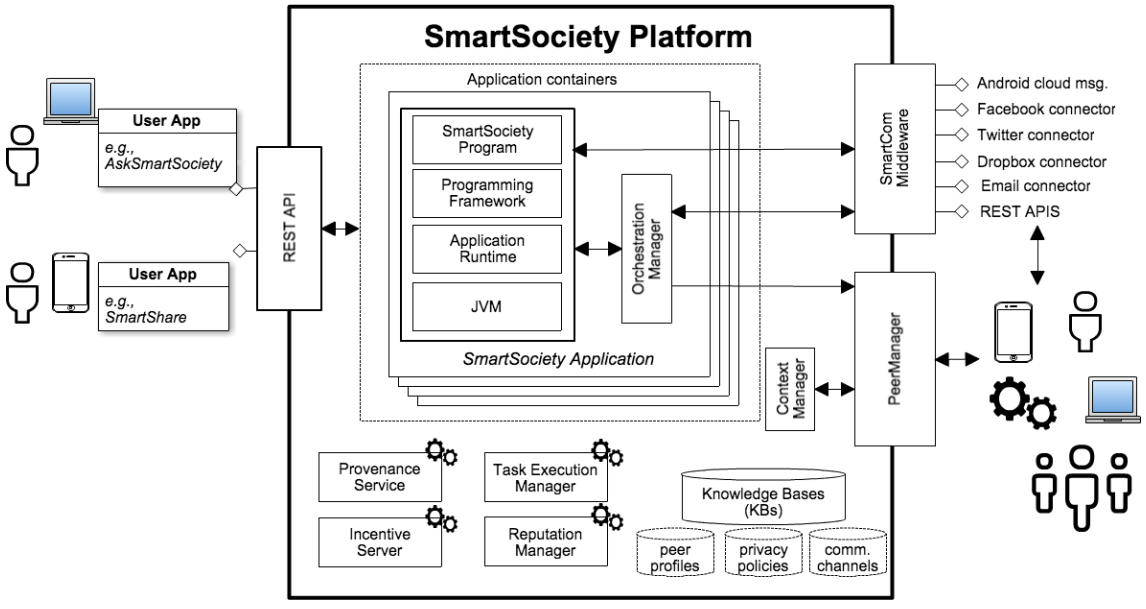

Figure 1: SmartSociety platform users and architecture.

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 [5]. 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 [6]. 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 [6].

- **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 [7]. 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 [8]. 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 [9]. 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 [10].
- **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 [5] 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 [6] 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 [6]. 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 [12]. 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 [12] 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 with 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. More details can be found in [4].

3.8.2 Implementation

The CM is implemented in Java.

3.8.3 Interfaces, Endpoints and Resources Exposed

Message information is available through REST APIs:

- **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. If there is already such a message information, it will be overridden with this information.

The Adapter REST Service exposes some Input adapters of SmartCom using REST:

- **DELETE /adapter/:id** Deletes an adapter with the given id.
- **POST /adapter/rest** Create a new rest input adapter that listens on a specific port and uri for incoming messages.
- **POST /adapter/email** Create a new email input adapter. The request includes subject, host, username/password, type, subtype fields.
- **POST adapter/dropbox** Create a new dropbox input adapter. The request includes dropboxKey, dropboxFolder, fileName, type, subtype and conversationId fields.

The other functionality is available through Java functions:

- **public Identifier send(Message message)** 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. The receiver of the message is defined within the message, it can be a peer or a collective.

- **public Identifier addRouting(RoutingRule rule)** 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.

- **public RoutingRule removeRouting(Identifier routeId)** Remove a previously defined routing rule identified by an Id. As soon as the method returnsthe routing rule will not be applied any more.

- **public Identifier addPushAdapter(InputPushAdapter adapter)** 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.

- **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.

- **public Identifier addPullAdapter(InputPullAdapter adapter, long interval, boolean deleteIfSuccessful)** 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.

- **public InputAdapter removeInputAdapter(Identifier adapterId)** 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.

- **public Identifier registerOutputAdapter(Class¡? extends OutputAdapter¿ adapter)** 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.

- **public void removeOutputAdapter(Identifier adapterId)** 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).

- **public Identifier registerNotificationCallback(NotificationCallback callback)** Register a notification callback that will be called if there are new input messages available.
- **public boolean unregisterNotificationCallback(Identifier callback)** Unregister a previously registered notification callback.

3.8.4 Repository

The CM code can be found at: `https://gitlab.com/smartsociety/SmartCom/`

3.9 Monitoring

3.9.1 Functionality and Features

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 and developers of SmartSociety-based CAS enabled applications and services. 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 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.

The implementation shipped with the SmartSociety toolkit [10] is based on the logstash open source framework[5], 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 elasticsearch for indexing and persistence. Aggregated and curated logs are stored in the elastic database and can be queried via standard interfaces, enabling technical supporting partners to develop their

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

own ad hoc monitoring dashboard or to integrate with legacy ones. The default option for SmartSociety is to use Kibana[6], a flexible dashboard which supports seamless integration with elasticsearch and presents basic, yet sufficient analytics functionality. 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.

3.9.4 Repository

Not applicable.

Tommaso: create project in gitlab and add dockerconfig to the repo

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

4 Expected Usage

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 [4], 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, execution plan composition, human participation negotiations, and finally the execution itself. These operations are provided by various SmartSociety platform components, which expose a set of APIs used by the PF. 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.

The CBT embeds a state machine managing transitions between states representing the various phases of the task lifecycle: provisioning, composition, negotiation and execution. An additional state, named continuous orchestration, is used to represent a state combining composition and negotiation under specific conditions [4].

We distinguish between two types of collectives:

- A Resident Collective (RC) denotes a stable group or category or peers based on the common properties. It is defined by a persistent PeerManager identifier, existing across multiple application executions, and possibly different applications.
- An Application-Based Collective (ABC) is a team or a group of people gathered around a concrete task. Differently from a resident collective, an ABC's lifecycle is managed exclusively by the SmartSociety application. Therefore, an ABC cannot be accessed outside of an application, and ceases to exist when the application's execution stops. An ABC can be created either implicitly (as intermediate products of different states of CBT execution) or explicitly (by using dedicated collective manipulation operators such as cloning a resident collective). An ABC has a lifetime equal to that of the application where it was created.

The application context (CTX) represents a particular instance of the SmartSociety application, and keeps all values, overall application state and metadata initialized for the

application run. The 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. The PF 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 specifies how CBT of a certain type can be built.

The CBT is instantiated through a CBTBuilder. Builders are registered at CTX initialization time and associated to a CBT type [4] 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:

```
CBT cbt = ctx.getCBTBuilder("RideSharingType")
.of(CollaborationType.OC)
.forInputCollective(c)
.forTaskRequest(t)
.withNegotiationArgs(myNegotiationArgs)
.build();
```

The basic method for checking the lifecycle state is

- **CBTState getCurrentState()** Returns the currents state.

It returns an enumeration CBTState with the active state of the CBT. Additional utility (pretty-print) methods for comparing the state of execution are detailed in [4].

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)

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.

This means that the transactions are by default explicitly controlled by the developer??

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).

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 isDone() returns true), and then retrieves its result.

- **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 overall execution in any state and transition directly to the final fail-state.
- **boolean isCancelled()** Returns true if CBT was canceled before it completed.

A CBT object exposes the following methods for fetching the ABCs created during CBT's 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 getNegotiables()** Returns the list of negotiable collectives.

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)** Creates and registers a collective with the PeerManager.
- **ResidentCollective createFromID(string ID, string to kind)** Creates a local 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.

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.
- **ABC join(Collective master, Collective slave, [string to kind)])** Creates an ABC instance, containing the union of peers from Collectives master and slave.

- **ABC complement(Collective master, Collective slave, [string to kind)])** Creates an ABC instance, containing the peers from Collective master after removing the peers present both in master and in slave. The resulting collective must be transformable into to 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 PeerManager persists the collective as RC.

The PF allows at the moment only a basic set of communication constructs, namely those for sending a message to a collective, and receiving responses from it:

- **void send(Message m)** Send a message to the collective. Does not wait for the sending to succeed or fail.
- **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)** Unregister a previously registered callback.

These methods are invokable on any Collective object.

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 [2]. In Table 1 we summarize the achievements of the platform development in terms of ability to meet the requirements identified by the Consortium.

As it can be seen, most of the requirements, but not all, have been met. In general, this reflects on the one hand a shift in focus

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.	Yes	The CM maintains the dynamic part of the profile
PP-2	Peer Profiles and Peer Profiling	Peer profiles data storage	Partially	See [6] for integration with PPL
PP-3	Peer Profiles and Peer Profiling	Peers can have multiple profiles	Partially	See [11, 6]
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	Yes	See previous sections of this document
PU-2	Platform Usability	Ease for developers to create and manage applications	Partially	Using PF interfaces
PU-3	Platform Usability	Support application development and deployment life-cycle automatically	Partially	Limited support for deployment
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 [13] and AskSmartSociety! demo.
HT-2	Platform Components and Interactions Heterogeneity	Platform components will run on a variety of heterogeneous devices	No	Runs 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 [14]
SEC-1	Privacy and Security	Access Control	Partially	See [6]
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 [6]
SEC-4	Privacy and Security	Peer Defined Usage Control Policies	No	
SEC-5	Privacy and Security	Secure Collection and Storage	Yes	See [6]
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	Partially	The codebase has not been optimized for performance

Table 1: Features of final version of the platform against requirements.

6 Conclusions

When back in 2012 the Consortium started working on the ideas and concepts that later on became the SmartSociety project the idea of a "platform for social computations" was meant to play a threefold role:

- To balance the flavour of R&D&I activities to be carried out in the project, making sure Consortium partners would not focus just on the abstract science of building and managing HDA-CASs, but would also consider constraints and barriers coming from the need to actually implement a working prototype thereof.
- To ensure consistency of the SmartSociety vision and outputs, acting as a single integration point for the whole Consortium activities [7].
- To respond to the need of having a computing infrastructure able to support HDA-CASs, to be used (first) by researchers for experimenting with CAS concepts and practises and (in the long term) to act as technology enabler for a new generation of ICT systems able to truly account for the human and social dimensions of computation in the real world.

This vision evolved over time as activities progressed and the Consortium gained more insight into the building blocks required to make a HDA-CASs working. The net output, which is presented in the deliverable at hand, should be understood as a prototypical implementation of a future computing infrastructure able to leverage hybrid, diversity-rich collectives to carry out complex computational tasks (possibly spanning the virtual and physical world).

The SmartSociety platform is — to a large extent — released under a permissive open source license[8]. While this is meant to ensure liveness of the software artifacts being developed, it should be coupled with an appropriate strategy for ensuring a coherent exploitation of the project's results. At the moment the overall platform can be considered somewhere between TRL-3 and TRL-4,[9] hence major efforts are needed to turn it into a

[7] This role is actually shared with WP1, which however focuses more on the ethical aspects and societal impacts of HDA-CASs.

[8] Only the PM and the CM are not publicly released, yet they APIs are, so that a developer skilled in the art could work out her bespoke implementation of the functionality and have it promptly integrated with the rest of the platform.

[9] It has to be noted that some components have actually higher TRL, for example SmartCOMis considered TRL-6.

product ready to hit the market. Yet, all the basic building blocks required to successfully execute hybrid computation at scale are there, so this is understood to represent a good starting point for anybody (scientist, startups and Web entrepreneurs, industries) active in the field.

At the time of writing an open source server model, similar to the one adopted by Prediction.io[10], seems to be the most viable solutions for ensuring a sustainable exploitation of the platform codebase: more details will be provided in the final project exploitation plan.

Something that is worth highlighting in this section is one issue that arose while discussing with WP1 team (see also [**?**] for a more extended discussion) on the ethical implications of some of the design choices that were taken by the project's technical team. When dealing with HDA-CASs indeed it turns out that design decisions in terms of enabling technology are at the center of three often conflicting forces: the technical aspects (with the technical team typically interested in shipping high-performance and spotless code), the business aspect (where the choice in terms of exploitation model and business plan can have a major impact on how the codebase develops) but, and this is pretty peculiar to HDA-CASs, also ethical ones (where adherence to a RRI approach does impact, often heavily, choices in terms of technology and exploitation model). To give a concrete example: WP1 put forward the need for a HDA-CASs to have a 'constitution' highlighting the values underpinning the CASs and the rights of all types of participants. Technically, this would mean that (referring to the PF primitives) when a CBT is instantiated it should have such constitution declared (which should be passed on to the CBTBuilder); in other words, some changes to the code are required. But if we decide to release the platform as an open source server, anybody could escape such steps by just tweaking the code, so this seems to also impact the business model rather significantly. At the same time if we decide to go for a PaaS model, which would allow us — as platform owners — to tightly control the constitution aspect, this would imply some major modifications to the codebase, which in turn would impact the financial perspectives in terms of launching a new venture. The tight entanglement of these three aspects (ethics, technology and business) appears to be probably the most challenging aspect of HDA-CASs, and certainly one of the major take-away of the project.

[10] `https://prediction.io/`

References

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

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

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

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

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

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

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

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

[10] I. Carreras, D. Miorandi, and T. Schiavinotto, "Second platform prototype and validation," SmartSociety FP7 Project Deliverable D8.2, Dec. 2014.

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

[12] M. Rovatsos, "Final models and validation," SmartSociety FP7 Project Deliverable D2.4, Jan. 2016.

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

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