Project Architecture

Deliverable 1.1 (version 1)

Université Lyon 1 - LIRIS

07 March 2015

Project ASAWoO

Adaptive Supervision of Avatar / Object Links for the Web of Objects

Grant Agreement: ANR-13-INFR-0012-04

Abstract

The deliverable has for objective to globally define the ASAWoO architecture. First, it defines the vocabulary, before describing the avatar architecture as well as the global view of avatar communities.

Contents

1 Introduction

In this deliverable, we present the specification of the global ASAWoO architecture. We first introduce our vocabulary (avatar, WoT application, functionality and capability), before describing our avatar architecture and its components, their role and operation (retrieval of basic object capabilities, functionality discovery, filtering and orchestration, code deployment, object collaboration and avatar interaction). We then introduce the notion of community and describe how avatars interact in communities.

2 Vocabulary

- **Avatar :** piece of software that extends a physical object
- **WoT Application (WoTApp) :** software component that has a GUI and offers a functionality to a user. An application can be available or not depending on the environment. The WoT application depends on the functionalities that the objects present in the environment offer.
- **Functionality :** high level features that avatars can expose on the Web and can be viewed as the interfaces that capabilities implement. Functionalities are utilized in WoT applications. There are three ways for implementing a functionality: directly with a capability exposed by an object or a service endpoint, or indirectly through a combination of several functionalities that are themselves implemented. The functionality class has a isComposedOf property that allows for describing complex functionalities that are composed of other functionalities. It also has an isImplementedBy property that describes how a functionality is realized with a capability.
- **Capability :** the physical action that an object can realize. Capability instances provide an implementation for a functionality. We defined three subclasses for the Capability class : ActuatorCapability, ProcessingCapability and SensorCapability, that respectively denote physical actions, data processing, and detection.

3 Avatar architecture

The avatar architecture is called the “WoT Runtime Environment”, as depicted in Fig 1. It is composed of a framework in which are plugged a set of components called managers, which are grouped in modules. Each manager takes charge of a specific concern and interacts with other components using a specific API[1]. The core module of the avatar architecture deploys the components into the framework via the “Component Deployment Manager” and furnishes low-level components that other managers require to perform caching and reasoning tasks. Each logical component of the avatar architecture can be executed either on the object or in the cloud. Avatars belong to a WoT infrastructure that is restricted to the local network for privacy reasons. However, the infrastructure enables inter-avatar communication as well as limited interactions with the external Web.
At initialization time, the “Capability Manager” of the “Local Functionality Module” queries the “Appliance Manager” to retrieve the basic capabilities that the object can perform. The “Appliance Manager” relies on drivers configured with the “Appliance Configuration Manager” and loaded via the “Appliance Driver Manager” to interact with the object.
Then, the “Local Functionality Manager” semantically discovers the functionalities that the object can achieve using these capabilities. To do so, it queries the internal reasoner of the avatar that reasons about the functionality ontology. Some functionalities are complex and involve several functionalities to be fully realized. In such case, there is a need for a specific language to describe the orchestration of functionalities. One possible solution is to use the SCXML language, however, it is possible to rely on a different language to describe complex functionalities depending on their needs. The discovered functionalities are filtered using the “Context Manager” and “Privacy Manager”, in order not to expose unsafe or unrealizable functionalities.

[1]Application Programming Interface

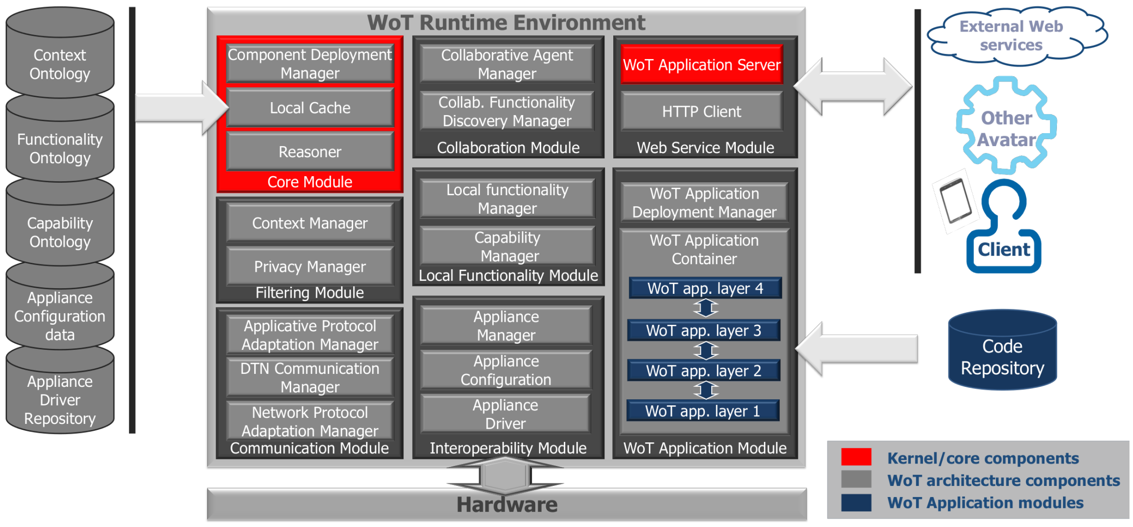


Figure 1: Avatar Architecture

The filtered functionalities are exposed as services to users and other avatars by the WoT application server. When a service is invoked, the local orchestration engine embedded in the avatar triggers the invocation of services based on the description of the functionality orchestration. For each “atomic” functionality (that is realized by a capability), the actual implementation of the capability is specific to each object and stored in an application code repository. The application code that implements the functionalities is executed in a second internal framework, called the “WoT Application Container”. The code is dynamically deployed from the code repository into the framework at execution time.
In the same fashion, the “Collaborative Functionality Discovery Manager” semantically discovers the functionalities in which the object can be involved but require collaboration with other objects. It is queried by the “Collaborative Agent Manager”, that interacts with other avatars through the Web service module.
The “WoT Application Server” component exposes available functionalities as REST resources. Each avatar thus uses the HTTP client module to lookup or invoke avatar resources in the WoT infrastructure, or query external Web services. The avatar also uses the Interoperability and Communication modules to respectively connect to the object and communicate with it using the most optimal network protocols. At the end of this document are the different interaction diagrams inside the avatar.

4 Avatar communities

Avatars on the Web form communities. Communities enable avatars to communicate with each others. We identify two parameters that demarcate the boundaries of communities.

- **geolocalisation :** avatars that represent physical appliances that interact together and provide physical functionalities if they are located at the same place
- **functionality :** avatars that provide location-independent functionalities such as calculation services can be located anywhere and can be made available in all the communities

We propose a federated structure for avatar communities, where the avatars located in the same place can directly interact together, and where requests to additional functionalities can be progressively redirected to wider areas until the required functionality is found. Avatars collaborate to provide users with WoT applications.
Interactions between avatars can lead to exhibit collective behaviours. We can concretely reuse all the works exhibited in multiagent systems in the context of these avatar: an avatar is an autonomous agent.

To enable collective behaviours, avatars must first discover which functionalities they can fulfil cooperatively. This will be done by requesting the application functionalities ontology. Then they must negotiate with one another to plan how to achieve these functionalities. This will require both a negotiation model and a communication protocol. This task will then be divided in three tasks:

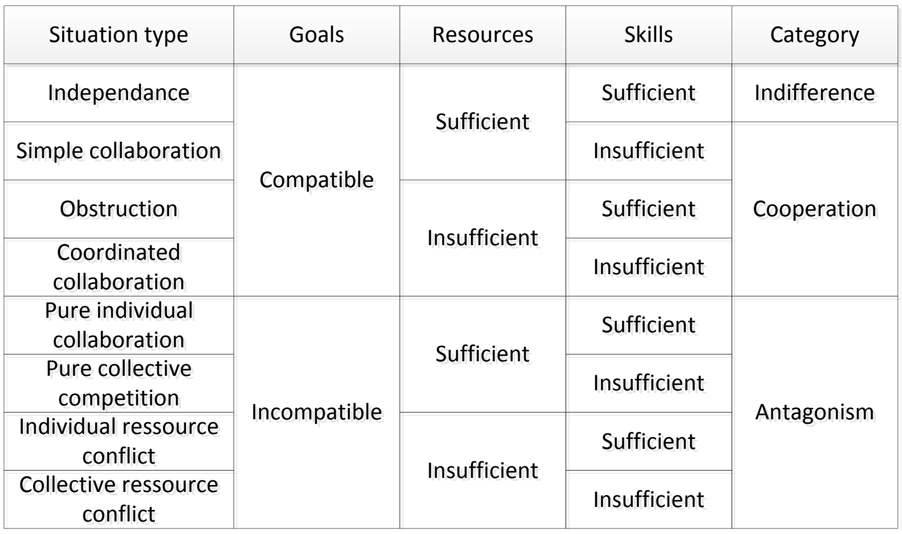
1. Enable complex tasks requests. An avatar has a functionalities management module. It may to respond to requests regarding functionalities that overcome the possibilities of the object. Technically, it consists in generating and sending to the functionalities ontology SPARQL requests equivalent to "what can I almost do with" and containing the set of local functionalities already discovered. The response to this query will be a set of super-functionalities, ordered by increasing number of missing functionalities in the input parameters and by the semantic distance between them. Indeed, we assume that when missing functionalities are close in the functionalities ontology, they are more likely to be available on the same object.

2. Inter-avatar cooperation model. It is necessary to define the core of the negotiation process between avatars, in order to identify functionalities that can be performed cooperatively. It will be based on a multiagent interaction model, so that avatars will be designed as autonomous agents that can use the context model to self-adapt their behaviour to contextual changes and pro-actively reorganize the set of exposed single and multi-object functionalities.

3. Inter-avatar communication as semantic Web services. Then, it will encapsulate the inter-avatar communication messages into semantic Web service. The challenge here is to reach semantic interoperability between various types of devices throughout standardized inter-avatars communication protocols. Languages and technologies from the semantic Web have been studied for the past few years as they allow describing a domain knowledge vocabulary in a semantics-explicit and machine-interpretable way, using an ontology description schema. The commonly adopted standard language for building ontologies is OWL, which is built on top of RDF and RDF-Schema. These advances have been exploited on the Web, in particular with semantic annotations relying on RDFa , microformats or microdata annotations. The envisioned format for exposing avatar functionalities in semantic Web services and "semantified" Web pages is currently HTML + RDFa.

Avatar are software entities evolving in an environment that they can partially perceive and in which they acts. They are endowed with autonomous behaviours and has objectives. Autonomy is the main concept in the agent issue: it is the ability of agents to control their actions and their internal states. The autonomy of agents implies no centralized control [6].
An avatar community can be seen as a multiagent system. It is set of avatars situated in a common environment, which interact and attempt to reach a set of goals. Through these interactions a global behaviour, more intelligent than the sum of the local intelligence of avatars, can emerge.The emergence paradigm deals with the non programmed and irreversible sudden appearance of phenomena in a system confirming that "the whole is more than the sum of each part". It is one of the expressions of collective intelligence [2].
Avatars can evolves different type of global behaviours (fig. 2). In this work, we assume the avatars want to reach the user requirements disregarding their own local goals (energy consumption reduction etc.). We then focus on cooperation and more precisely on simple collaboration situations. Our approach enable to change these assumptions but, in this case, we have to adapt individual behaviours of avatars.
Building a collaborative system requires to inspect different of its aspects [3]. In multiagent system, designers pay attention to:

- the *environment* deals with elements necessary for the multiagent system realization such as the perception of this environment and the actions one can do on it
- the *organizations* allows to order agent groups in organizations determined according to their roles.
- the *interaction* includes all elements at stake for structuring the external interactions among the agents like agent communication language and interaction protocols.
- the *agents* gathers all elements for defining and constructing these entities. It concerns the agent's know-how and knowledge, its model and its architecture.



Situation type	Goals	Resources	Skills	Category
Independance	Compatible	Sufficient	Sufficient	Indifference
Simple collaboration			Insufficient	Cooperation
Obstruction		Insufficient	Sufficient	
Coordinated collaboration			Insufficient	
Pure individual collaboration	Incompatible	Sufficient	Sufficient	Antagonism
Pure collective competition			Insufficient	
Individual ressource conflict		Insufficient	Sufficient	
Collective ressource conflict			Insufficient	

Figure 2: Classification of interaction situation [4]

These notions can be considered from a global (system centred) or a local (agent centred) point of view. Building the multiagent systems consists often into integrate these different aspects in agent architectures. Communication enable avatar to exchange information. They are at the origin of interactions and social organization of avatars. An interaction protocol governs relations between avatars to meet the global goals of the multi-avatar system. However avatars are autonomous: they can initiate, sustain, and detach themselves from these relationships. Considering an agent which accepts to use an interaction protocol, it have to accept the associated semantic too. When the agent is in a given state of the interaction, it can receive one message of a finite known set. Its answer will be a message of a pre-determinate set too.
They are a lot of works on interaction protocols. The contract net protocol (fig. 3) is an example of interesting interaction protocol for cooperative problem among agents. It provides a solution for the so-called connection problem: finding an appropriate agent to work on a given task. It is the good candidate for your problem. Considering an avatar *a* which require a functionality. The contract net interaction protocol is implemented into four steps:

1. Announce: Avatar *a* initiated interaction to announce its requirement;
2. Proposition: Avatars which can perform the functionality send their proposals;
3. Decision: After analysis of the received proposals, *a* choose the proposition of *b* which seem be the best solution;
4. Contract: The contract between *a* and *b* is engaged.

Organization is seen as a division of tasks and a distribution of roles. At each role is associated a behaviour. Here, roles and tasks coming from the scenario analysis. But another organization can occurs. In fact, our approach where where socially situated avatar may depend on one another to achieve their own goals can lead to the notion of dependence networks. Considering an avatar *a* which gives a proposal to an initiator avatar *b*. *a* can be initiator of another interaction because he requires a task to make the proposal. Dependence theory [1, 5] provides a nice framework to model such phenomena.

References

[1] CASTELFRANCHI, C., CESTA, A., CONTE, R., AND MICELI, M. Foundations for interaction: The dependency theory. In *Advances in Artificial Intelligence, Third Congress of the Italian Association for*

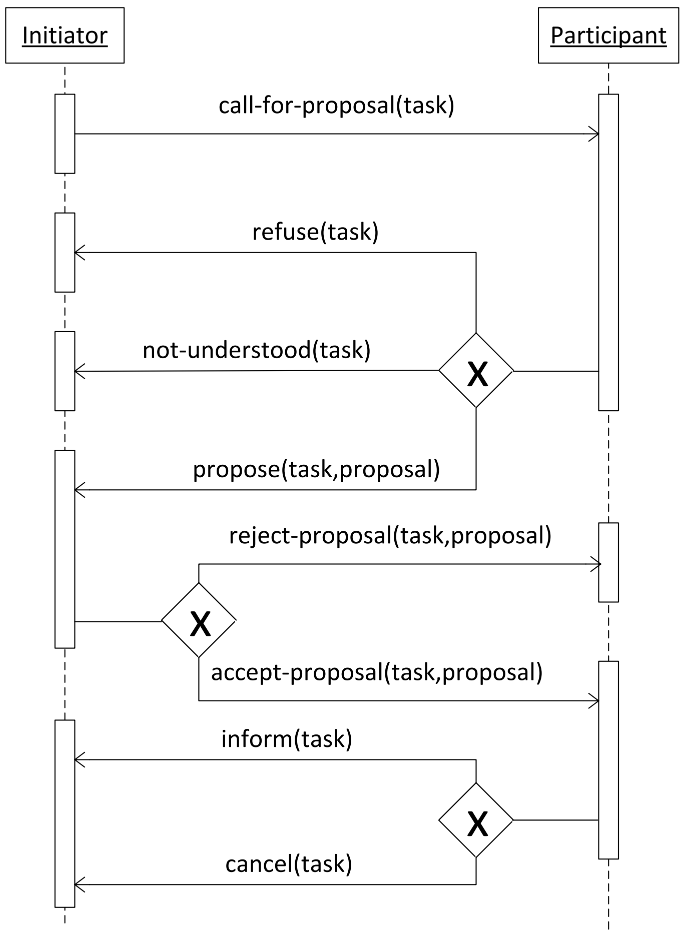


Figure 3: AUML representation of the Contract Net protocol

*Artificial Intelligence, AI*IA'93, Torino, Italy, October 26-28, 1993, Proceedings* (1993), vol. 728 of *Lecture Notes in Computer Science*, Springer, pp. 59–64.

[2] DEGUET, J., DEMAZEAU, Y., AND MAGNIN, L. Elements about the emergence issue: A survey of emergence definitions. *Complexus 3*, 1-3 (2006), 24–31.

[3] DEMAZEAU, Y. From interactions to collective behavior in agent-based systems. In *European Conference on Cognitive Science* (Saint-Malo France, 1995).

[4] FERBER, J. *Multi-Agent Systems: An Introduction to Distributed Artifical Intelligence*. Paperback, 1999.

[5] SICHMAN, J. S., CONTE, R., CASTELFRANCHI, C., AND DEMAZEAU, Y. A social reasoning mechanism based on dependence networks. In *ECAI* (1994), pp. 188–192.

[6] WOOLDRIDGE, M.-J. Intelligent agents. In *Multiagent systems : A modern approach to Distributed Artificial Intelligence* (1999), G. Weiss, Ed., MIT Press.

5 Interaction diagrams for avatar components

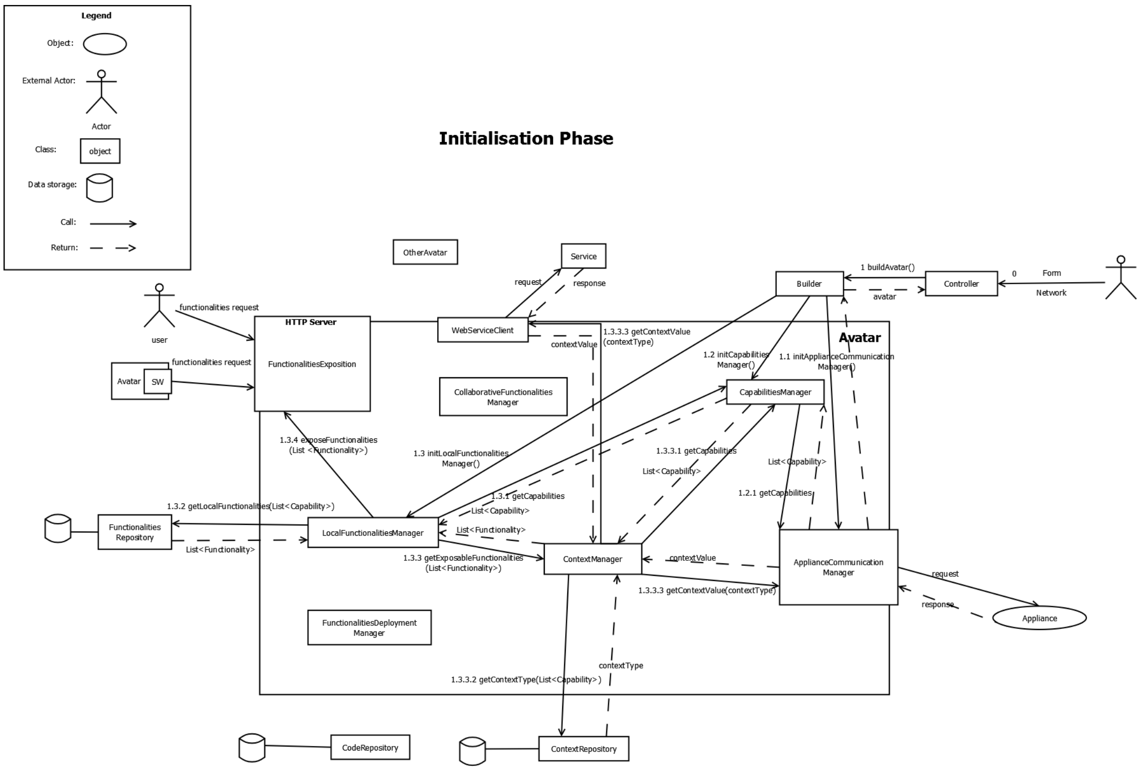

Legend
Object:
External Actor:
Actor
Class:
object
Data storage:
Call:
Return:
Initialisation Phase
OtherAvatar
Service
request
response
WebServiceClient
Builder
1 buildAvatar()
Controller
0
Form
Network
avatar
user
functionalities request
HTTP Server
FunctionalitiesExposition
Avatar
SW
functionalities request
1.3.3.3 getContextValue (contextType)
contextValue
Avatar
1.2 initCapabilities Manager()
1.1 initApplianceCommunication Manager()
CollaborativeFunctionalities Manager
CapabilitiesManager
1.3.4 exposeFunctionalities (List <Functionality>)
1.3 initLocalFunctionalities Manager()
1.3.3.1 getCapabilities
List<Capability>
List<Capability>
1.2.1 getCapabilities
1.3.1 getCapabilities
List<Capability>
1.3.2 getLocalFunctionalities(List<Capability>)
Functionalities Repository
LocalFunctionalitiesManager
List<Functionality>
List<Functionality>
1.3.3 getExposableFunctionalities (List<Functionality>)
ContextManager
contextValue
ApplianceCommunication Manager
1.3.3.3 getContextValue(contextType)
request
response
Appliance
FunctionalitiesDeployment Manager
contextType
1.3.3.2 getContextType(List<Capability>)
CodeRepository
ContextRepository

Figure 4: Initialization phase

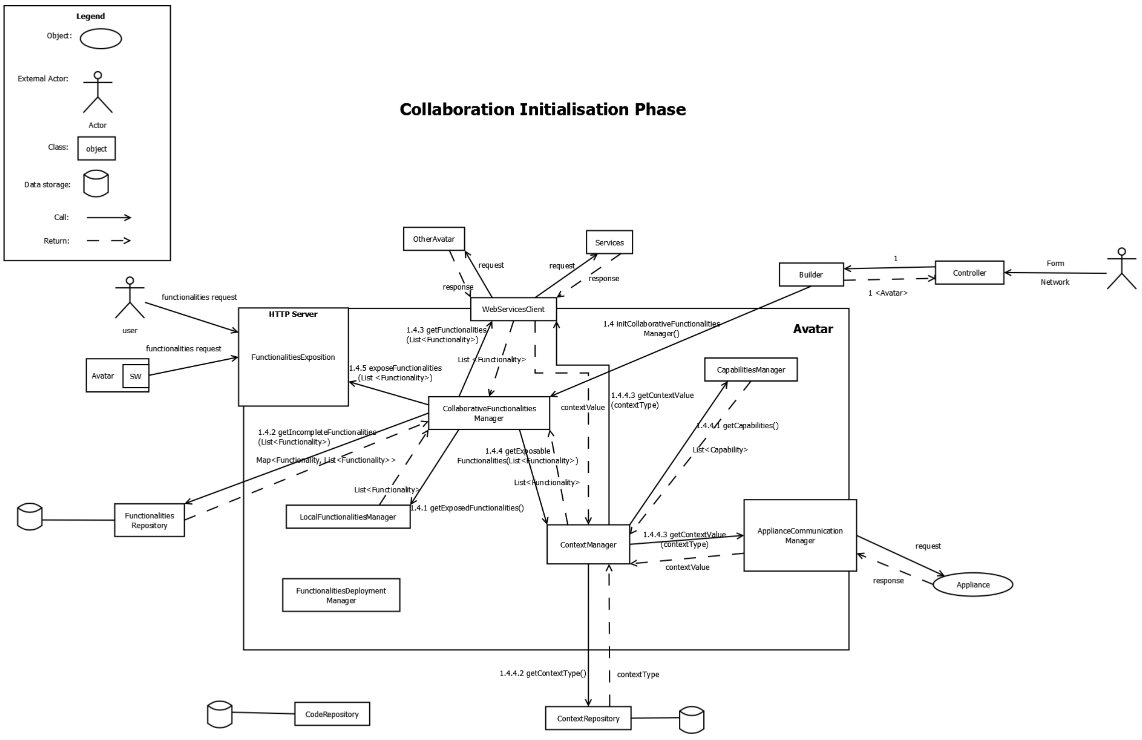

Legend
Object:
External Actor:
Actor
Class:
object
Data storage:
Call:
Return:
Collaboration Initialisation Phase
OtherAvatar
Services
request
request
response
response
Builder
1
Controller
Form
Network
1 <Avatar>
user
functionalities request
HTTP Server
FunctionalitiesExposition
WebServicesClient
1.4.3 getFunctionalities (List<Functionality>)
1.4 initCollaborativeFunctionalities Manager()
Avatar
functionalities request
Avatar
SW
List <Functionality>
1.4.5 exposeFunctionalities (List <Functionality>)
CapabilitiesManager
CollaborativeFunctionalities Manager
contextValue
1.4.4.3 getContextValue (contextType)
1.4.2 getIncompleteFunctionalities (List<Functionality>)
1.4.4.1 getCapabilities()
Map<Functionality, List<Functionality>>
1.4.4 getExposable Functionalities(List<Functionality>)
List<Capability>
List<Functionality>
List<Functionality>
1.4.1 getExposedFunctionalities()
Functionalities Repository
LocalFunctionalitiesManager
ContextManager
1.4.4.3 getContextValue (contextType)
ApplianceCommunication Manager
request
contextValue
response
Appliance
FunctionalitiesDeployment Manager
1.4.4.2 getContextType()
contextType
CodeRepository
ContextRepository

Figure 5: Collaboration initialization phase

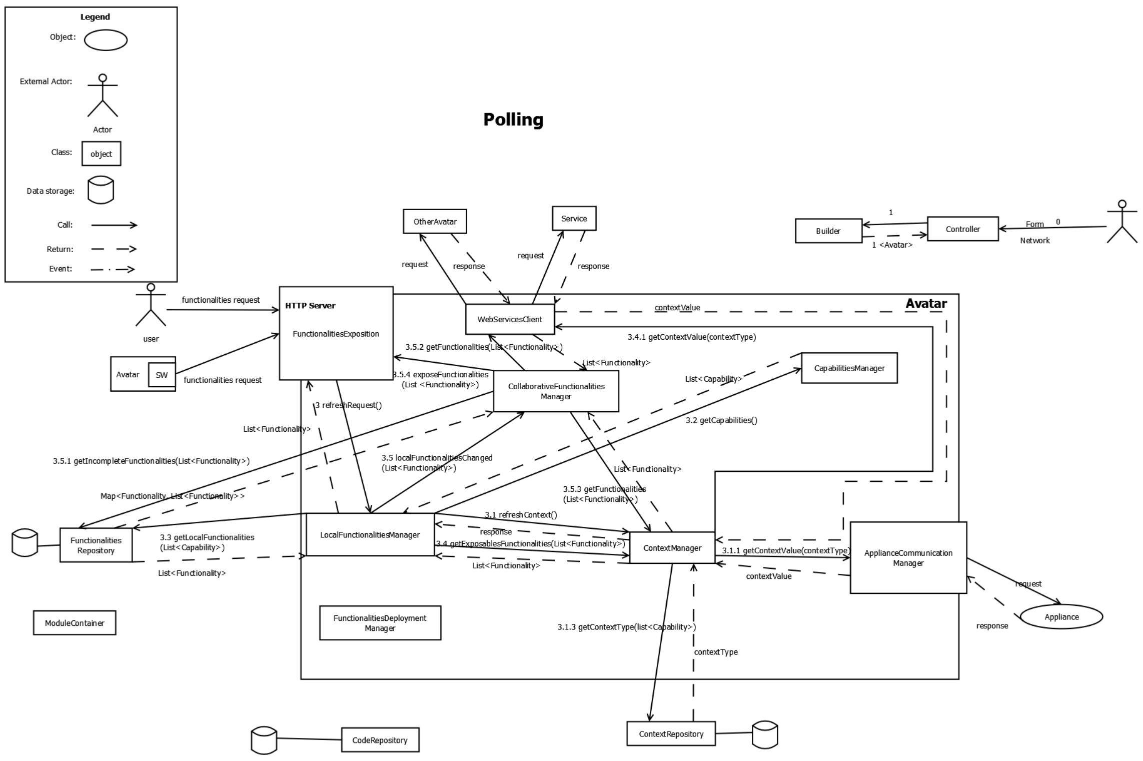

Legend
Object:
External Actor:
Actor
Class:
object
Data storage:
Call:
Return:
Event:
Polling
OtherAvatar
Service
Builder
Controller
Form
Network
request
response
user
functionalities request
HTTP Server
FunctionalitiesExposition
WebServicesClient
contextValue
Avatar
3.4.1 getContextValue(contextType)
3.5.2 getFunctionalities(List<Functionality>)
List<Functionality>
CapabilitiesManager
3.5.4 exposeFunctionalities (List <Functionality>)
CollaborativeFunctionalities Manager
List<Capability>
3 refreshRequest()
3.2 getCapabilities()
List<Functionality>
3.5.1 getIncompleteFunctionalities(List<Functionality>)
3.5 localFunctionalitiesChanged (List<Functionality>)
Map<Functionality, List<Functionality>>
3.5.3 getFunctionalities (List<Functionality>)
3.1 refreshContext()
Functionalities Repository
3.3 getLocalFunctionalities (List<Capability>)
LocalFunctionalitiesManager
response
3.4 getExposablesFunctionalities(List<Functionality>)
ContextManager
3.1.1 getContextValue(contextType)
ApplianceCommunication Manager
List<Functionality>
contextValue
request
Appliance
response
ModuleContainer
FunctionalitiesDeployment Manager
3.1.3 getContextType(list<Capability>)
contextType
CodeRepository
ContextRepository

Figure 6: Event Polling

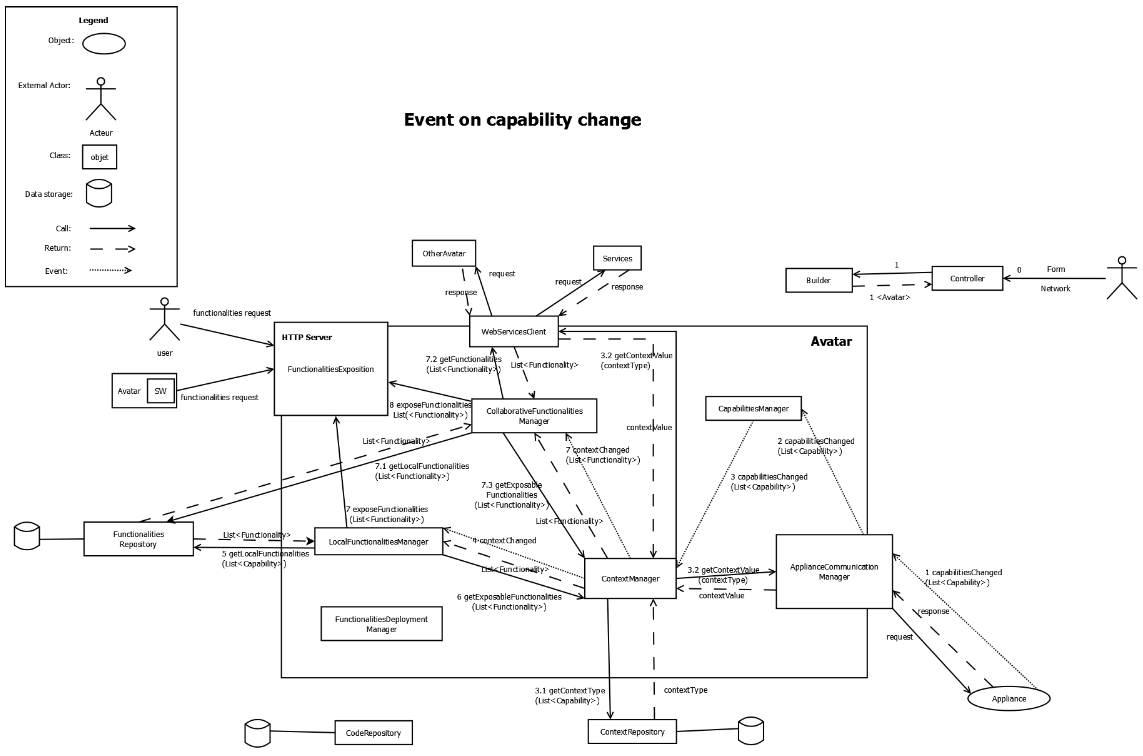

Legend
Object:
External Actor:
Acteur
Class:
objet
Data storage:
Call:
Return:
Event:
Event on capability change
OtherAvatar
Services
request
response
Builder
Controller
Form
Network
1 <Avatar>
user
functionalities request
HTTP Server
FunctionalitiesExposition
WebServicesClient
Avatar
7.2 getFunctionalities (List<Functionality>)
List<Functionality>
3.2 getContextValue (contextType)
8 exposeFunctionalities (List(<Functionality>)
CollaborativeFunctionalities Manager
CapabilitiesManager
contextValue
List<Functionality>
7 contextChanged (List<Functionality>)
2 capabilitiesChanged (List<Capability>)
7.1 getLocalFunctionalities (List<Functionality>)
3 capabilitiesChanged (List<Capability>)
7.3 getExposable Functionalities (List<Functionality>)
7 exposeFunctionalities (List<Functionality>)
List<Functionality>
Functionalities Repository
List<Functionality>
LocalFunctionalitiesManager
4 contextChanged
5 getLocalFunctionalities (List<Capability>)
List<Functionality>
3.2 getContextValue (contextType)
ApplianceCommunication Manager
1 capabilitiesChanged (List<Capability>)
ContextManager
contextValue
6 getExposableFunctionalities (List<Functionality>)
response
request
FunctionalitiesDeployment Manager
Appliance
3.1 getContextType (List<Capability>)
contextType
CodeRepository
ContextRepository

Figure 7: Event Capability Change

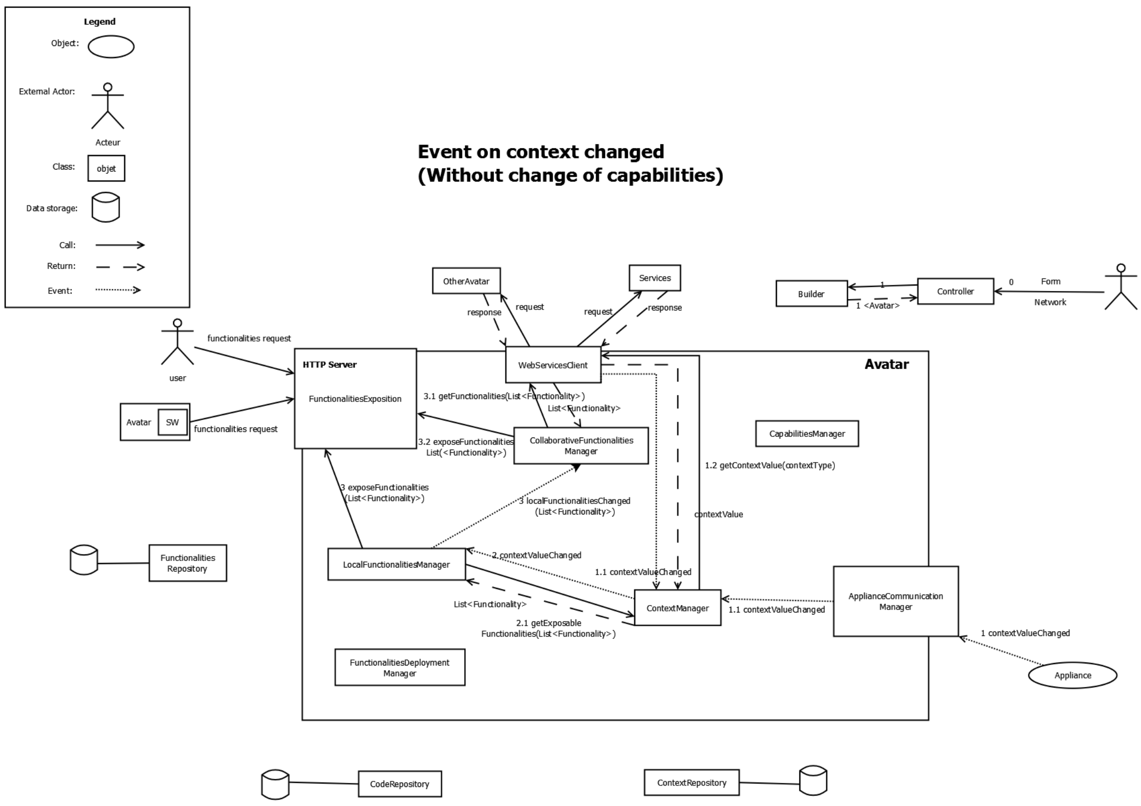


Figure 8: Event Context Change

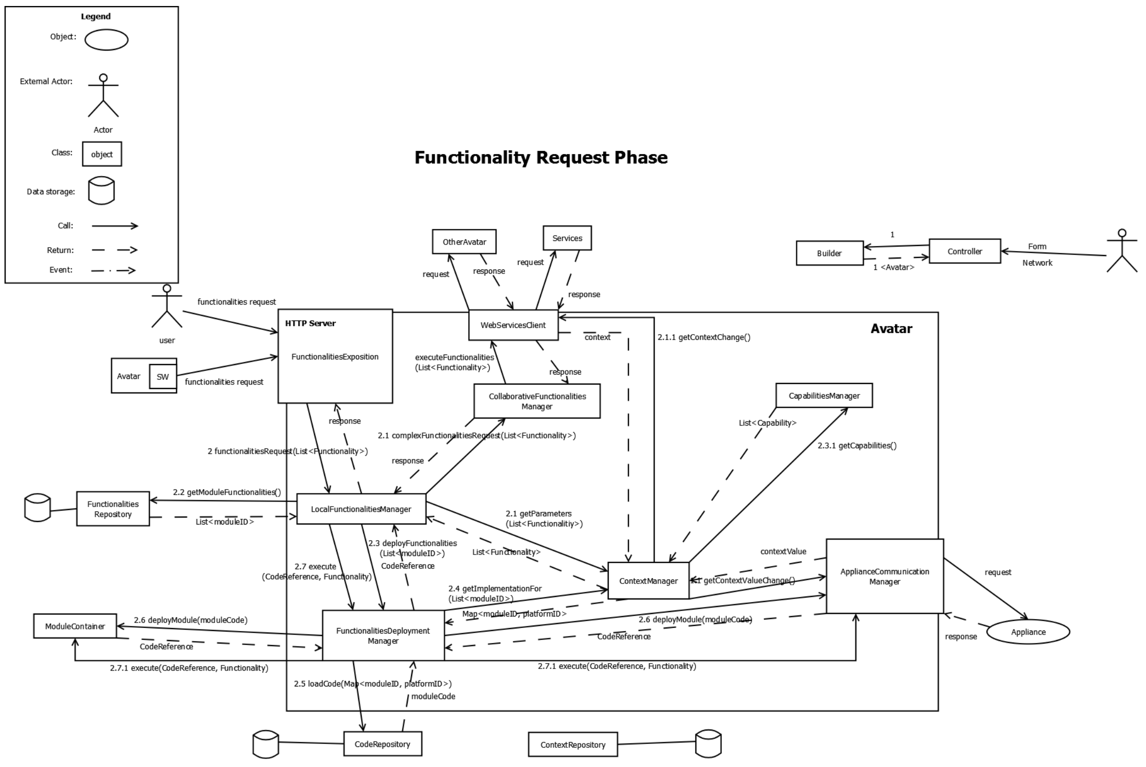


Figure 9: Functionality Request Phase