Project no.:	**610917**
Project full title:	**Sustainable and reliable robotics for part handling in manufacturing**
Project Acronym:	**STAMINA**
Deliverable no.:	**4.1.2**
Title of the document:	**Report on the implementation of SOA for robot fleets**

Contractual Date of Delivery to the CEC:	**31.12.2015**
Actual Date of Delivery to the CEC:	**21.12.2015**
Organisation name of lead contractor for this deliverable:	**INESC Porto**
Author(s):	**César Toscano, Germano Veiga, Matthew Crosby, Ron Petrick**
Work package contributing to the deliverable:	**WP4**
Nature:	**R**
Version:	**1.0**
Total number of pages:	**19**
Start date of project:	**01.10.2013**
Duration:	**42 months – 31.03.2017**

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 610917

Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

This deliverable reports the implementation of the STAMINA service-oriented architecture.

Document History

Version	Date	Author (Unit)	Description
0.1	01-12-2015	César Toscano (INESC Porto)	Creation of document.
0.2	15-12-2015	César Toscano (INESC Porto)	Submitted to WP4 for analysis and feedback.
0.3	18-12-2015	Volker Krüger (AAU)	Revised/validated whole document.
1.0	19-12-2015	César Toscano (INESC Porto)	Generated final version.

Table of Contents

Figures

Tables

Glossary

Term	Definition
API	Application Program Interface, specifies the visible interface of a given software element.
CRUD	Create Retrieve Update Delete
EIS	Enterprise Information Systems.
ERP	Enterprise Resource Planning.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
JSON	JavaScript Object Notation, syntax for describing data objects in java script language.
MES	Management Execution System.
REST	REpresentational State Transfer.
ROS	Robot Operating System.
RPC	Remote Procedure Call.
UML	Unified Modelling Language.
URI	Uniform Resource Identifier.
Web-Service	Software system designed to support interoperable machine-to-machine interaction over a network (World Wide Web Consortium – W3C definition). A Web-Service has a specific interface, described in the XML and WSDL languages.

1 Introduction

1.1 Scope and Objectives

This document constitutes the third outcome of Task 4.1 “Service-Oriented Interfaces for Robot Fleets”, which aims at providing the robot fleets with service oriented interfaces allowing them to communicate with the interface layers of the Enterprise Information System (EIS) at PSA. The objective of the task is to develop those interfaces in a sustainable way, taking into account standards, compliance and flexibility of use.

STAMINA developments are based on a well-developed and community supported robot operating systems, namely the Robot Operating System (ROS). This system has its own internal messaging mechanisms that are designed with performance in mind but are not interoperable, without the loss of loose coupling with the EIS systems. On the other hand, the diversity and complexity of the EIS systems also limit a consistent integration of the same solution across several plants (either from the same company or not). As an example, the PSA EIS is proprietary and therefore all developments must take into account any future generalizations to that system.

This document follows deliverable D4.1.1, version 2 (STAMINA consortium, 2015a) and reports the implementation of the service-oriented architecture therein presented. The reader is advised to read that report in order to have a detailed description and justification of the STAMINA service-oriented architecture. Complementary information is presented in deliverable D4.2.3 (STAMINA consortium, 2015b) as the reader finds here a comprehensive characterization of the activities performed by the Logistic Planner component.

1.2 Document organization

After this introductory chapter, the report contains the main chapter of the document which describes how the STAMINA service-oriented architecture was implemented in terms of technology and software artefacts. This is followed by a concluding chapter where the next steps are also identified.

1.3 References

STAMINA consortium, 2015a. “Deliverable D1.1.1 Specification of SOA for robot fleets”, version 2, 29-04-2015. The abstract is the following:

> “This deliverable provides the second specification of the service interfaces for STAMINA robot fleets. The major characteristics of service-oriented architectures are outlined, followed by the description of STAMINA’s service-oriented architecture: major components and their interactions represented in the form of services. This is complemented by the description of a concrete software implementation to be used in the second test sprint.”

STAMINA consortium, 2015b. “Deliverable D4.2.3 Conversion logic and high level operations scheduling”, version 1.0, 30-09-2015. The abstract is the following:

> “This deliverable provides the specification of the logistic planning activities performed by the Logistic Planner component.”

STAMINA consortium, 2015c. “Deliverable D1.3.6 STAMINA test and evaluation report M22”, version 1.0, 31-07-2015. The abstract is the following:

> “This deliverable provides a report on the tests performed during the second test sprint. This performance report will provide a structured feedback to the following R&D phase and prepare the next test sprint.”

STAMINA consortium, 2015d. “Deliverable D3.2 Revised report on skill architecture”, version 1.0, 28-05-2015. The abstract is the following:

> “This deliverable presents the revised and final architecture of the skill based system SkiROS, one of the core components in the STAMINA project. The deliverable explains the skills, architecture, and integration on the robot and the link to the higher level planning system.”

STAMINA consortium, 2015e. “Deliverable D4.3.1 Architecture Description for the Mission Planning Subsystem”, version 1.0, December-2015. The abstract is the following:

> “This deliverable reports on the architecture for the Mission Planner subsystem that is situated between the EIS and the individual robots in the STAMINA system. The Mission Planner is responsible for producing mission assignments for the individual robots in the fleet using information on kitting orders, robot skills, and the state of the operating environment provided by the Logistic Planner. Generated missions are returned to the Logistic Planner which ultimately relays them to the individual robots for execution.”

2 Implementation of the Service-oriented architecture for robot fleets

2.1 Information flow and services

Figure 1 identifies the components that comprise the STAMINA system and the corresponding information flow. Colours are used to differentiate the two main categories of information elements: logistic world model information are represented in green and order management information in blue. At the bottom level, each STAMINA robot, apart from all the hardware components, is managed by three ICT elements: a **Task Manager**, containing a **Task Planner** inside it, and several **Skill Managers**, which encapsulate and hide all the details concerning the execution of the actions performed by the robot. These three elements operate on top of the robot's operating system (ROS), running as ROS Nodes, in the most autonomous way as possible. At the top level, the **Logistic Planner** and the **Mission Planner** have general visibility over the robot fleet. The Mission Planner runs also as a ROS node while the Logistic Planner runs as a server in the Java virtual machine environment which is supported by both Windows and Linux. The term EIS represents all the external systems integrated into STAMINA (e.g. ERP and MES). The **World Model Consistency Checker** is a ROS node running also within the robot.

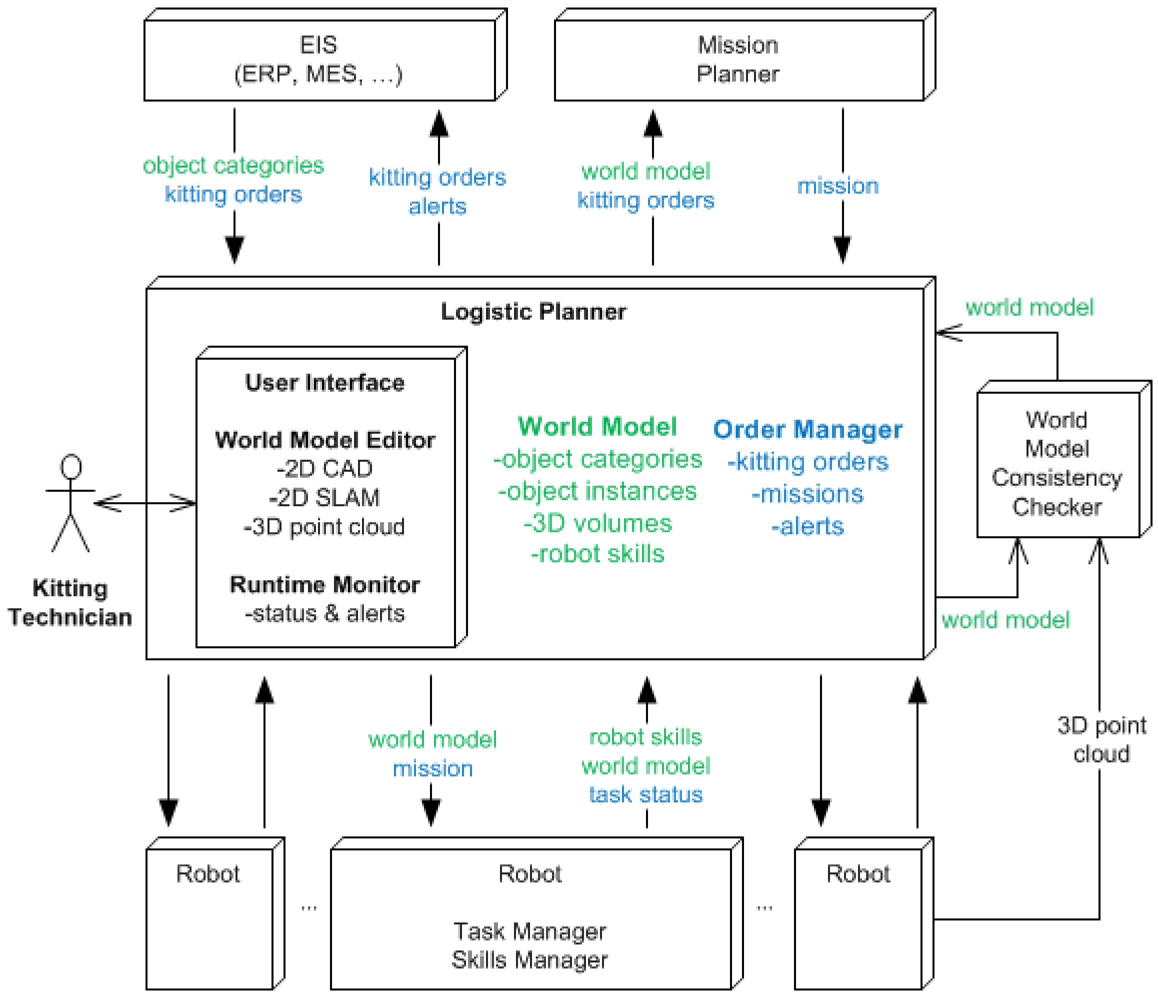


Figure 1 – High level system architecture (components and information flow).

Figure 2 identify the services that support the information flow depicted in Figure 1. The arrows in these two diagrams identify the direction of service invocation. Following sections describe in detail the implementation of these services.

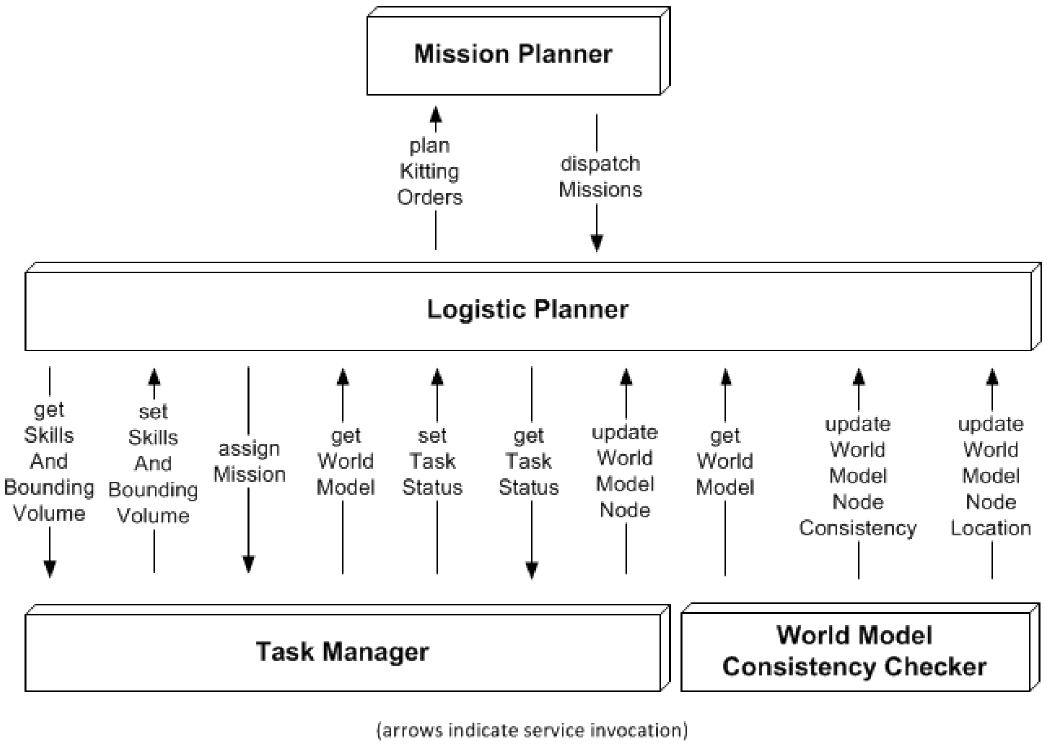

Mission Planner
plan Kitting Orders
dispatch Missions
Logistic Planner
get Skills And Bounding Volume
set Skills And Bounding Volume
assign Mission
get World Model
set Task Status
get Task Status
update World Model Node
get World Model
update World Model Node Consistency
update World Model Node Location
Task Manager
World Model Consistency Checker
(arrows indicate service invocation)

Figure 2 – STAMINA components' services.

For a detailed description of these two pictures, the reader should consult the following reports: STAMINA consortium, 2015a, STAMINA consortium, 2015b, STAMINA consortium, 2015d and STAMINA consortium, 2015e.

As described in the following sections, the services supporting the interaction of the Logistic Planner with both the Mission Planner and the Task Manager were implemented on top of ROS, thus making use of their communication mechanisms (STAMINA consortium, 2015a describes the ROS communication model). The interaction between the Logistic Planner and the World Model Consistency Checker followed a different implementation and is based on the REST model (details described below).

2.2 ROS implementation

A first implementation of the service-oriented architecture described above was completed in time for the second test sprint where the services were tested and validated successfully (see STAMINA consortium, 2015c). This included all the services depicted in Figure 2, with the exception of the *setSkillsAndBoundingVolume* and *updateWorldModelNode* which will be tested on the third test spring (2016).

Ubuntu 14.04.1 constituted the operating system hosting the STAMINA components (Logistic Planner, Mission Planner, Task Manager and Skill Manager), as depicted in Figure 3. All of these components are available as ROS nodes[1] (ROS Indigo release) and programmed in the C++ language. The exception is the Logistic Planner. As this component aims to integrate with the external EIS systems, and setup actions are needed from the person responsible for the kitting zone so that logistic information is provided to the remaining STAMINA components, more adequate programming tools were selected and used to implement the component: Java standard edition supports its server side implementation while the client side is supported by JavaScript and an HTML 5 environment, so that the Logistic Planner's user interface can run on standard internet browsers.

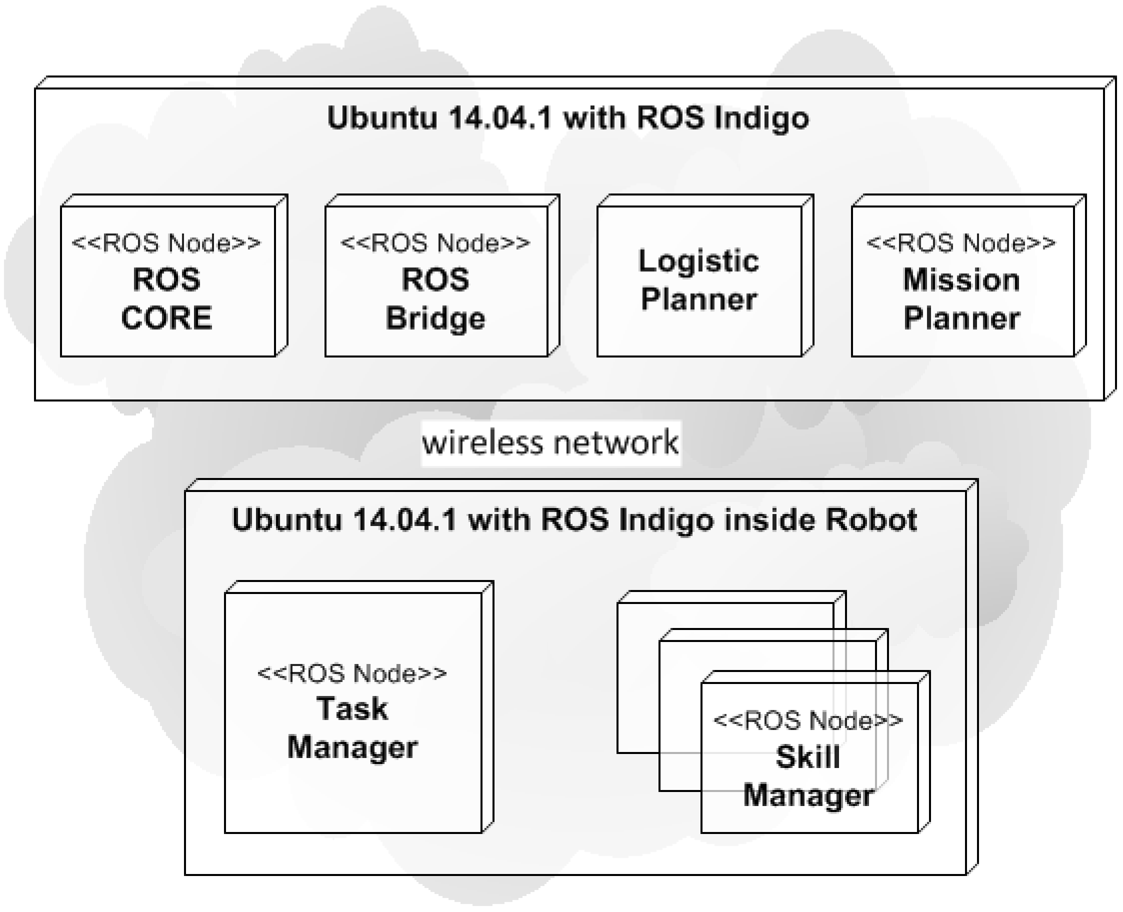

Ubuntu 14.04.1 with ROS Indigo
<<ROS Node>> ROS CORE
<<ROS Node>> ROS Bridge
Logistic Planner
<<ROS Node>> Mission Planner
wireless network
Ubuntu 14.04.1 with ROS Indigo inside Robot
<<ROS Node>> Task Manager
<<ROS Node>> Skill Manager

Figure 3 – STAMINA Components and ROS nodes (UML deployment diagram).

In this instance of the STAMINA architecture, services correspond to ROS services and topics. Access to/from the Logistic Planner is made through the ROS Bridge. The ROS Bridge is a ROS node that facilitates communications from the ROS world (Mission Planner, Task Manager, Skill Manager) to other entities running in different computational environments. This bidirectional communication channel is implemented as an HTML 5 web socket. Thus, the Logistic Planner communicates with all remaining STAMINA components through the ROS Bridge. Service invocations made by the Logistic Planner are sent to the ROS Bridge and from there to the targeted ROS node (for example, to the Mission Planner). The corresponding response from the service is sent to the ROS Bridge and then to the Logistic Planner. Inversely, a proxy of the services provided by the Logistic Planner is available at the ROS Bridge so that an invocation of this service will be translated into the invocation of the real service available at the Logistic Planner.

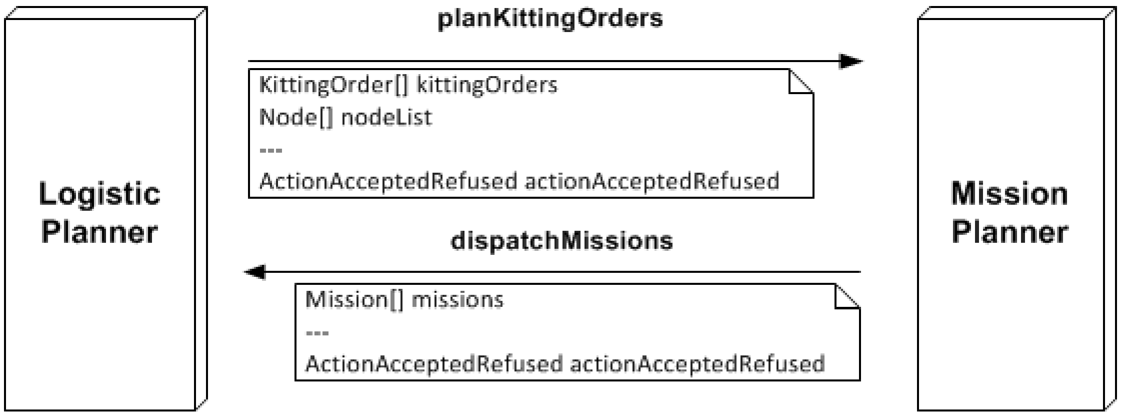
[1] A ROS node is a process that performs a given computation. A ROS based system is comprised of several ROS nodes that communicate through services and topics.

In terms of communication, internal transactions between the Logistic Planner and the ROS Bridge are hidden: the ROS Nodes available in the network are not aware of the presence of the ROS Bridge. Additionally, the exchanged ROS messages are translated by the ROS Bridge into the JSON format. The Logistic Planner sends and receives JSON messages and the ROS Bridge translates these messages into a ROS-specific format.

2.2.1 Services

Most of the STAMINA services are implemented as ROS Services: this is the case where an action is requested and a synchronous response is required stating the acceptance or refusal of the action.

Figure 4 identifies all the ROS services supporting the interaction between the Logistic Planner and the Mission Planner. Each arrow indicates an invocation of a service and the rectangle below the arrow specifies the request and response data items (under ROS convention, the fields above the three hyphens represent the request message and the ones below the response message).


Logistic Planner
planKittingOrders
KittingOrder[] kittingOrders
Node[] nodeList

ActionAcceptedRefused actionAcceptedRefused
dispatchMissions
Mission[] missions

ActionAcceptedRefused actionAcceptedRefused
Mission Planner

Figure 4 – ROS services and messages: Logistic Planner - Mission Planner interactions.

A detailed description of the Mission Planner is reported on Deliverable 4.3.1 (STAMINA consortium, 2015e). Here the reader will find information on how missions are assigned to the individual robots in the fleet using information on kitting orders, robot skills, and the state of the operating environment provided by the Logistic Planner (logistic world model).

Table 1 specifies the name and type of each service (used at runtime to locate the provider of the service).

Table 1 – ROS service names: Logistic Planner - Mission Planner interactions.

planKittingOrders LogisticPlanner requests the MissionPlanner to plan a set of kitting orders.	
ROS service name:	*/stamina/MissionPlanner_PlanKittingOrders*
ROS service type:	*stamina/MissionPlanner_PlanKittingOrders*
dispatchMissions MissionPlanner sends to LogisticPlanner a set of planned missions.	
ROS service name:	*/stamina/LogisticPlanner_DispatchMissions*
ROS service type:	*stamina/LogisticPlanner_DispatchMissions*

Following the same notation, Figure 5 identifies all of the ROS services supporting the interaction between the Logistic Planner and the Task Manager. Each arrow indicates an invocation of a service and the rectangle below the arrow specifies the request and response data items.

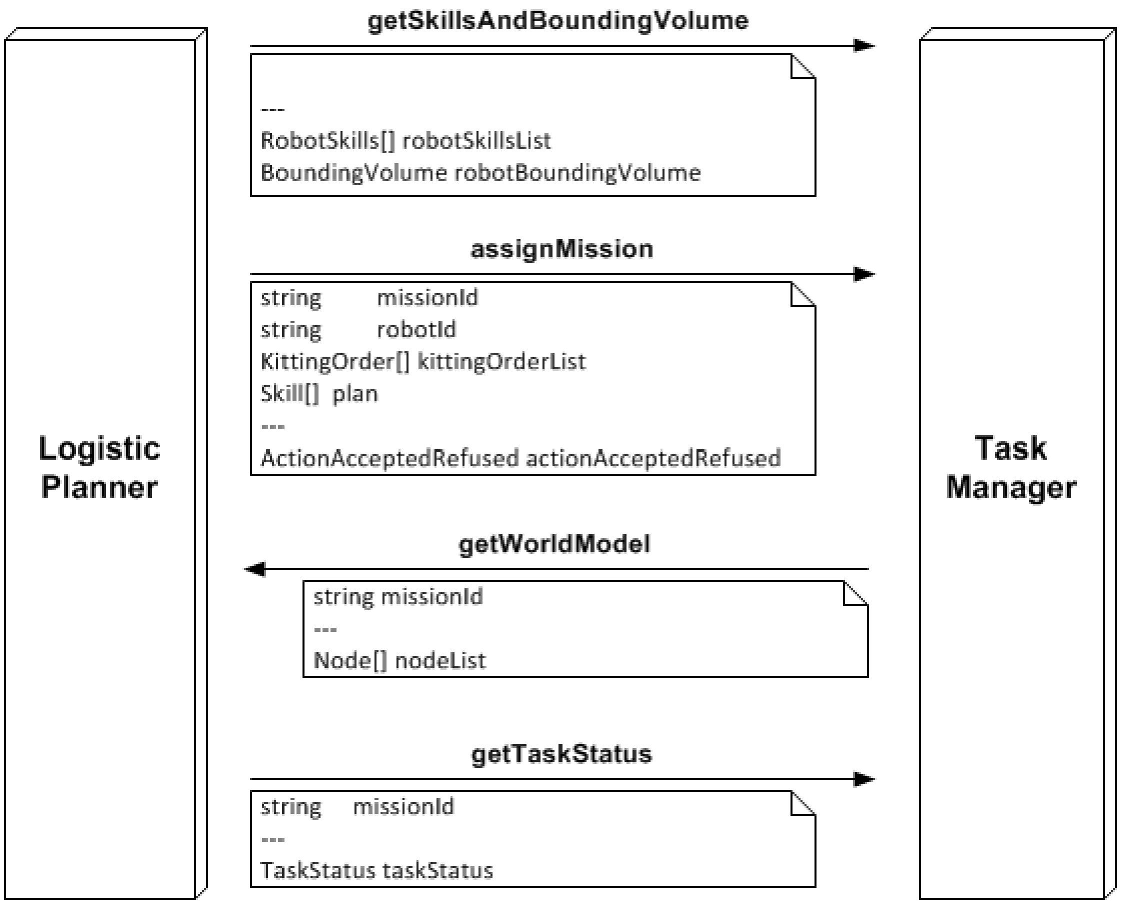


Figure 5 – ROS services and messages: Logistic Planner - Task Manager interactions.

A detailed description of the Task Manager is reported on Deliverable 3.2 (STAMINA consortium, 2015d). Here the reader will find information on the skills architecture and how they are implemented and retrieved from the individual robots in the fleet.

Table 2 specifies the name and type of each service (used at runtime to locate the provider of the service).

Table 2 – ROS service names: Logistic Planner – Task Manager interactions.

getSkillsAndBoundingVolume LogisticPlanner requests to TaskManager for a list of all implemented skills and the robot's bounding volume.	
ROS service name:	*/skiros_task_manager/ProvideSkillsAndBoundingVolume*
ROS service type:	*skiros_task_manager/ProvideSkillsAndBoundingVolume*
assignMission LogisticPlanner requests the TaskManager to start executing a given planned mission.	
ROS service name:	*/skiros_task_manager/AssignMission*
ROS service type:	*skiros_task_manager/AssignMission*

getWorldModel TaskManager requests LogisticPlanner to provide the world model for a given mission.	
ROS service name:	*/stamina/LogisticPlanner_ProvideWorldModel*
ROS service type:	*stamina/LogisticPlanner_ProvideWorldModel*
getTaskStatus LogisticPlanner requests to TaskManager for the actual execution status of a mission.	
ROS service name:	*/skiros_task_manager/ProvideTaskStatus*
ROS service type:	*skiros_task_manager/ProvideTaskStatus*

2.2.2 Topics

Some of the STAMINA services are implemented as ROS Topics: this is the case where the caller of a service needs to issue a notification, but there is no need to have a synchronous response.

Figure 6 identifies all of the ROS topics supporting the interaction between the Logistic Planner and the Task Manager. Each arrow indicates an invocation of a service and the rectangle below the arrow specifies the request data items. These invocations correspond to a topic generated by the Task Manager (no response is generated by the Logistic Planner).

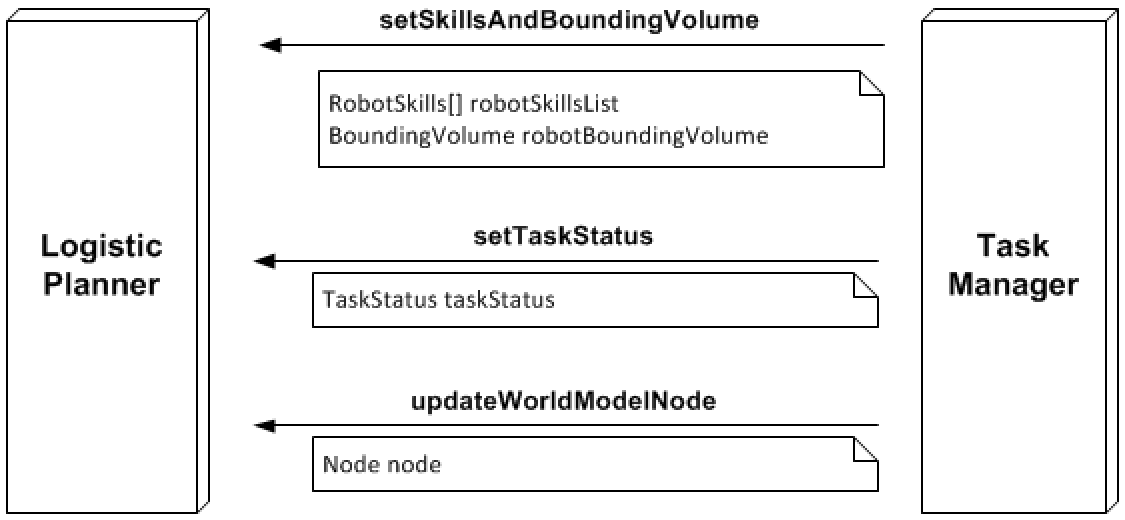


Figure 6 – ROS topics: Logistic Planner - Task Manager interactions.

Table 3 specifies the name and type of each topic (used at runtime to locate the provider of the service).

Table 3 – ROS topic names: Logistic Planner – Task Manager interactions.

setTaskStatus TaskManager sends to LogisticPlanner the actual execution status of a mission.	
ROS topic name:	*/skiros_task_manager/TaskStatus*
ROS topic type:	*skiros_task_manager/TaskStatus*
ROS topic message:	*TaskStatus taskStatus*
setSkillsAndBoundingVolume TaskManager sends to LogisticPlanner same information as above.	

ROS topic name:	*/skiros_task_manager/SkillsAndBoundingVolume*
ROS topic type:	*skiros_task_manager/SkillsAndBoundingVolume*
ROS topic message:	*SkillsAndBoundingVolume robotSkillsAndBoundingVolume*

2.2.3 Data objects

The previous ROS services and topics use ROS messages containing data objects. The following table specifies all such data objects.

Table 4 – ROS data objects.

Object	*Type of data item*	*Name of data item*
RobotSkills	string	robotId
	Skill[]	skillList
Skill	string	skillName
	string	skillPDDL
	string[]	parameters_names
	string[]	parameters_values
ActionAcceptedRefused	string	accepted
	string	reason_of_refusal
Node	string	id
	string	name
	BoundingVolume	boundingVolume
	NodeProperty[]	nodeProperties
	string[]	childrenIdList
BoundingVolume	OriginLocation	originLocation
	OriginOrientation	originOrientation
	Point[]	polyLine
	float64	height
OriginLocation	float64	x
	float64	y
	float64	z
OriginOrientation	float64	r
	float64	p
	float64	y
Point	float64	x
	float64	y
	float64	z
NodeProperty	string	name
	string	value
	string	type

KittingOrder	string	id
	string	carSeqNumber
	string	kitId
	PartToPick[]	partsToPick
PartToPick	string	cellId
	string	partId
	uint8	quantity
Mission	string[]	kittingOrderIdList
	string	robotId
	Skill[]	plan
TaskStatus	string	missionId
	string	status
	time	when

2.2.4 Service registry

In this concrete service-oriented architecture implementation, the service registry is implemented by a special ROS node: the ROS Master. The ROS Master is responsible for keeping a record of the following:

- which ROS node is providing a specific service,
- which ROS node is publishing a specific topic, and
- which ROS node subscribed to a specific topic.

The first time a ROS node calls a ROS service, it contacts the Master node to determine which node in the network is providing that service. Subsequently, the invocation node will communicate directly with the node providing the given service. This behaviour assumes that the service provider initially registered the service on the Master node. De-registration of the service should occur as soon as the service provider disables access to the service.

In a similar way, before publishing a topic message for the first time, the publisher node must register itself at the ROS Master. All nodes interested in receiving any messages belonging to that topic must register their interest at the ROS Master. Subsequent creation of topic messages uses this information so that all subscriber nodes receive those topic messages.

2.3 REST implementation

The World Model Consistency Checker component aims to validate parts of the logistic world model initially created in the Logistic Planner. For that purpose, the logistic world model is periodically retrieved and compared with the reality sensed in the kitting area. As traffic may be considerable, a lighter communication model was selected: the REST model, which will be described in the next section.

2.3.1 Type of RESTfull services

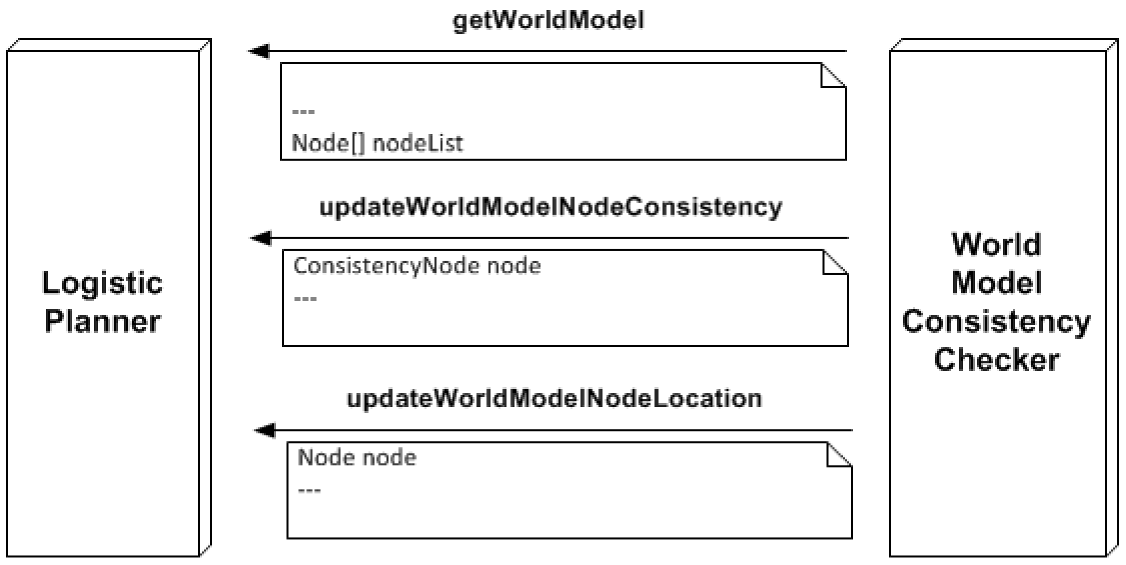
Technologically, the services supporting the interaction Logistic Planner – World Model Consistency Checker are implemented as RESTfull web-services over the HTTP protocol and by making use of the JSON data model. The REpresentational State Transfer (REST) architectural style models services as resources identified by a URI (Uniform Resource Identifier) that have a state and representation. The representation of a resource includes hyperlinks to other resources in the Internet so that a client consuming a REST service has access to those hyperlinked resources. The state and behaviour of a resource is made available through a set of standard methods (GET, POST, DELETE, PUT, HEAD, OPTIONS) defined in the HTTP protocol. Following the best practices of the Future Internet, the first four types of service invocations are used in STAMINA as follows:

- In order to retrieve the current representation of a given entity (e.g., a Node), the HTTP GET message is used. An identifier parameter of the given entity may be present in the request for the cases where this is required. The HTTP 200 response message contains in its body the representation of the requested entity in the JSON format. For example, in order to get the current representation of a given Node, one invokes the URL http://hostname/worldmodel_node/13570, where *hostname* specifies the host implementing the RESTfull service and 13570 identifies a given Node. The body of the HTTP response contains the data elements representing the requested Node in the form of a JSON data element.
- In order to update the current representation of a given entity (e.g., a *Node*), the *HTTP POST* message is used. An identifier parameter of the given entity may be present in the request for the cases where this is required. The body of the POST message contains the new representation of the identified resource. The *HTTP 200* response message simply indicates the successful execution of the action. For example, in order to update the current representation of a given *Node*, one invokes the URL *http://hostname/worldmodel_node/node1092*, where *hostname* specifies the host implementing the RESfull service and *node1092* identifies a given *Node*. The body of the *HTTP POST* message contain the data elements representing the *Node* in the form of a JSON data element.
- In order to remove an existing resource (e.g., a *Node*), the *HTTP DELETE* message is used. An identifier parameter of the given entity may be present in the request for the cases where this is required. The *HTTP 200* response message simply indicates the successful execution of the action. For example, in order to remove the current representation of a given *Node*, one invokes the URL *http://hostname/worldmodel_node/13570*, where *hostname* specifies the host implementing the RESfull service and *13570* identifies the *Node* to be removed.
- In order to create a resource (e.g., a *Node*), the *HTTP PUT* message is used. The body of the *PUT* message contains the representation of the resource to be created. The *HTTP 200* response message indicates the successful execution of the action and do contain in its body the initial data elements required to create the resource. For example, in order to create a new *Node*, one invokes the URL *http://hostname/worldmodel_node*, where *hostname* specifies the host implementing the RESfull service. The data elements that will be used in the creation of the *Node* are placed in the body of the *PUT* message as a JSON data element. The body of the *HTTP 200* response message contains the identifier of the *Node* just created.

Exceptions during the execution of a RESTfull service are returned as appropriate HTTP messages (depending on the condition that raised the exception): *400* “BAD_REQUEST”, *401* “UNAUTHORIZED”, *403* “FORBIDDEN”, *404* “NOT_FOUND”, and *500* “INTERNAL_SERVER_ERROR”.

2.3.2 Services

Figure 7 identifies the REST services supporting the interaction between the Logistic Planner and the World Model Consistency Checker. Each arrow indicates an invocation of a service and the rectangle below the arrow specifies the request and response data items (the ROS convention is followed, where the fields above the three hyphens represent the request message and the ones below the response message).


getWorldModel

Node[] nodeList
updateWorldModelNodeConsistency
ConsistencyNode node
---
updateWorldModelNodeLocation
Node node

Logistic Planner
World Model Consistency Checker

Figure 7 – REST services: Logistic Planner – World Model Consistency Checker interactions.

Table 5 lists all the REST interaction services (Logistic Planner – World Model Consistency Checker) designed and implemented in software so far. For each service, the first column in the table identifies its symbolic name, the second column specifies the HTTP message used to invoke the service and the third column identifies the URI pattern used to invoke the service.

Table 5 – List of EIS integration services

Service	HTTP Method	Resource URI pattern	Provider
getWorldModel	GET	/worldmodel_complete	LogisticPlanner
updateWorldModelNodeConsistency	POST	/worldmodel_consistencynode/:id	LogisticPlanner
updateWorldModelNodeLocation	POST	/worldmodel_locationnode/:id	LogisticPlanner
updateWorldModelNode	POST	/worldmodel_node/:id	LogisticPlanner

The *getWorldModel* service allows the World Model Consistency Checker to periodically retrieve the current logistic world model from the Logistic Planner. This comprises an instance of the *WorldModel* class identified in Figure 8, where Node objects are hierarchically linked as described in STAMINA consortium, 2015a. Subsequently, and as a result of the real-time analysis performed by the World Model Consistency Checker, the following three services allow Logistic Planner to be informed of the following situations (threshold levels are used to compare the sensed reality with the corresponding cybernetic representation):

- A given cell in a rack is occupied by an object with a volume that is consistent with the information contained in the logistic world model (e.g., the cell contains a small box);
- A given cell in a rack doesn’t contain a volume as stated by the logistic world model;

- A given rack or large box is located in a place that is different from the one stated in the logistic world model.

2.3.3 Data objects

Figure 8 comprise the UML class diagram of all the data objects that are used in the REST-based interaction services. Each class of objects is represented by a rectangle with a title (e.g., *Node*) and a set of attributes (e.g. *id* and *name*). Composition associations are represented in the diagram.

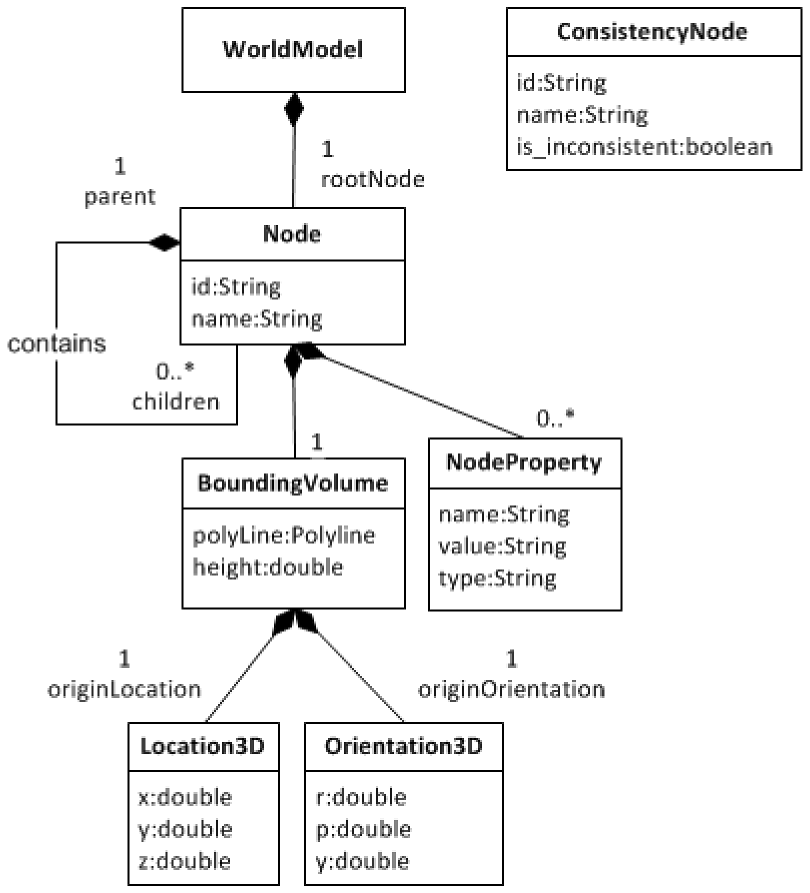


Figure 8 – REST data objects: Logistic Planner – World Model Consistency Checker interactions.

The world model is represented by a set of *Node* objects. Each such object comprises a *BoundingVolume* object, describing the volume occupied by the corresponding physical object and its three-dimensional location and orientation in space, and comprises also a set of *NodeProperty* objects. Each *NodeProperty* object describes a name/type/value triplet.

3 Conclusion and next steps

The STAMINA service-oriented architecture was first specified in Deliverable 4.1.1, version 1.0. Seven months later, the same report was revised and submitted in April-2015. This specification is aligned with the implementation that was tested and validated successfully in the second test sprint at the PSA Rennes factory (March-May, 2015). This was reported in Deliverable D1.3.6, published in 31-07-2015.

Software implementation of the STAMINA service-oriented architecture was mainly achieved through ROS technology. ROS services and topics were the instruments supporting the interaction between the STAMINA components: Logistic Planner, Mission Planner and Task Manager. However, implementation of the latter two components was achieved through the C++ programming language (native language of ROS) while Java was the language (and computing environment) used to program the Logistic Planner. The usage of the ROS Bridge component (available in the ROS official distribution) ensured a transparent communication between the Logistic Planner and the other two components: in the perspective of any ROS node (e.g. Mission Planner and Task Manager), communication is realized as ROS services and topics. Details about the implementation of ROS services on the Mission Planner are provided by Deliverable 4.1.3.

For supporting the third test sprint (starting Feb-2016), two additional ROS services were specified (updateWorldModelNode and setSkillsAndBoundingVolume). Additionally, and in order to ensure the consistency of the logistic world model, a first specification and implementation of the interaction between the Logistic Planner and the World Model Consistency Checker already started. In this particular case, and in order to avoid performance issues, the REST communication model was selected. In this context, the services are being implemented as RESTfull services.