Sustainable and reliable robotics for part handling in manufacturing

Project no.:	**610917**
Project full title:	**Sustainable and reliable robotics for part handling in manufacturing**
Project Acronym:	**STAMINA**
Deliverable no.:	**4.1.1**
Title of the document:	**Specification of SOA for robot fleets**

Contractual Date of Delivery to the CEC:	**30.09.2014**
Actual Date of Delivery to the CEC:	**30.09.2014**
Organisation name of lead contractor for this deliverable:	**P06 INESC Porto**
Author(s):	**Arnaud Chazoule, César Toscano, Eric Fauré, Germano Veiga, Matthew Crosby, Ron Petrick**
Work package contributing to the deliverable:	**WP4**
Nature:	**R**
Version:	**1.0**
Total number of pages:	**17**
Start date of project:	**01.10.2013**
Duration:	**42 months – 31.03.2017**

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
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 provides the first specification of the service interfaces for STAMINA robot fleets. The major characteristics of a service-oriented architecture are described followed by the identification of an abstract service-oriented architecture for STAMINA. This is achieved in a technology neutral way enabling later decisions to be made by the project's Consortium.

Document History

Version	Date	Author (Unit)	Description
0.2	23-09-2014	César Toscano (INESC Porto)	Draft for evaluation in the consortium.
0.3	29-09-2014	César Toscano (INESC Porto)	Updated draft after evaluation in the consortium.
1.0	29-09-2014	César Toscano (INESC Porto)	Prepared version to be published.

Table of Contents

Figures

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.
FAC	”Fiche Accompagnment”, created by PSA MES system, identifies the kit to be built in the kitting zone.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
JSON	JavaScript Object Notation, syntax for describing data objects in java script language.
Large Volume	Type of packaging mainly used for material handling of large items/goods. Large Volume containers are usually handled by forklifts.
MES	Management Execution System.
ROS	Robot Operating System.
RPC	Remote Procedure Call
Small Boxes	Box-type packaging used for material handling of small items/goods. Small Boxes are usually stored in specific Kanban racks.
Takt Time	Maximal time to realise a mission or an assembly task (in this case maximal time to provide a kit).
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.
WSDL	Web Services Description/Definition Language
XML	Extensible Markup Language, syntax for describing structured data vocabularies.
XSD	XML Schema Definition

1 Introduction

1.1 Scope and Objectives

This document reports the first 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 the standards compliance and flexibility of use.

STAMINA developments are to be based on well-developed and community supported robot operating systems, namely the Robot Operating System (ROS). These systems have their internal messaging mechanisms that are designed with performance in mind but are not interoperable, without the loss of loose coupling with EIS systems. On the other hand the diversity and complexity of the EIS systems also limits 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 the developments must take into account the future needed generalisation.

This document is closely related with Deliverable D4.2.1 "Specification of the integration services with ERP systems and related sub-systems" but while this one focuses on the integration services with EIS, identifying all the PSA EIS elements and specifying a service-oriented integration mechanism (established by STAMINA's Logistic Planner), the present document goes beyond it and specifies the STAMINA service-oriented interfaces that are required to create missions, plan tasks and control their skills execution by the robot fleet.

1.2 Document organization

This document is organised as follows: after this introductory chapter 1, the basic concepts and design principles around the service oriented architecture style are presented in chapter 2 with the identification of supporting technology, namely SOAP-based and REST-based web services. Chapter 3 proceeds with a high level description of the ROS communication model which will be used for supporting communication between ROS specific elements in STAMINA system. Last chapter 4 proposes a service-oriented architecture for STAMINA system that will be subsequently detailed in the next test-sprint.

1.3 References

O'Brien et al. 2005. "Quality Attributes and Service-Oriented Architectures", Carnegie Mellon University / Software Engineering Institute, technical note CMU/SEI-2005-TN-014.

Bianco et al. 2007. "Evaluating a Service-Oriented Architecture", Carnegie Mellon University / Software Engineering Institute, technical report CMU/SEI-2007-TR-015.

Ganci et al. 2005. "Patterns: SOA Foundation Service Creation Scenario", IBM.

STAMINA consortium, 2014a. "Deliverable D1.1.1 First internal report on use case and experiment specification including evaluation criteria", version 1.0, 27-03-2014.

STAMINA consortium, 2014b. "Deliverable D1.2.1 Report on Concept, Architecture and Integration of STAMINA demonstrator (version M11)", version 1.0, 30-08-2014.

STAMINA consortium, 2014c. “Deliverable D3.1 Preliminary report on skill architecture”, version 1.0, 30-07-2014.

STAMINA consortium, 2014d. “A Preliminary Overview of Mission Planning in STAMINA”, version 1.2, 09-09-2014.

STAMINA consortium, 2014e. “Deliverable D4.2.1 Specification of the integration services with ERP systems and related sub-systems”, version 1.0, 30-09-2014.

2 Service-oriented architecture overview

This chapter aims to make an overview of a service-oriented architecture (SOA), by describing their key terms, components and supporting technology.

2.1 Definition

"Service" is the very first term used to describe a service-oriented architecture. It represents a reusable business task/function, encapsulating the functional units of an application (the processes supported by the functional unit) through a well-defined interface (the API) that is independent of any programming language (interface is implementation independent). Services are consumed (invoked) by other entities that could be client applications or services themselves.

"Service orientation" denotes a method of integrating business applications and supported processes as connected or composed services.

Given these two basic terms and taking a business perspective, a "service-oriented architecture" defines the set of connected and composed business services that capture the business design that an organization exposes internally and/or to its customers and business partners. At a lower level, i.e., at the implementation level, services are realised through computer artefacts, where technologies (e.g. Web services) are selected to fulfil the services. From an operational perspective, in a service-oriented architecture agreements between the consumers and the providers must be defined in order to define the quality of service, its cost model, etc.

As identified above, services can also act as consumers of other services, allowing one to have a service that integrates a set of other related services, achieving a "composite service".

2.2 Basic components

Service providers and consumers constitute the basic components in a SOA. By definition, a service provider may also act as a service consumer. In order to enable the establishment of a connection among two services (and possibly to establish composition of services), a third basic component is defined, the service registry, responsible for keeping the identification of all the provided services and their interface definitions available, on demand, to service consumers. The service registry is itself a service. Service providers use this service to publish the services they provide so that service consumers are consequently able to use the service registry to discover the services they are looking for. After the lookup of a service, the service consumer connects to the service provider in order to invoke the service.

2.3 Service-level design principles

A common set of service-level design principles most associated with service orientation are commonly accepted (O'Brian et all. 2005):

- Services are reusable – services are designed to support potential reuse.
- Services share a formal contract – a formal contract should be shared by service consumers and providers describing the terms of the information exchange and of any other service-related characteristics.
- Services are loosely coupled – services are to interact on a loosely coupled basis.

- Services abstract underlying logic - the service's description and formal contract is the only part of a service that is visible to service consumers, any required underlying logic should be invisible and irrelevant to the outside world.
- Services are compostable – in order to enable functionality at different levels of granularity, services may be used to define other services, thus promoting reusability and the creation of different levels of abstraction in the services themselves.
- Services are autonomous - The logic governed by a service resides within an explicit boundary. The service should have complete autonomy within this boundary and be not dependent on other services for the execution of this governance.
- Services are stateless – state information should not be managed by the service as this may impede the services' ability to remain loosely coupled. Statelessness should be maximised.
- Services are discoverable – the description of a service (i.e. its API) should be discovered by potential consumers so that they may be subsequently invoked.
- Services have a network-addressable interface – service consumers must be able to invoke a service through the network.
- Services are location transparent – consumers dynamically discover the location of a service by looking up a registry and not by initially using its network absolute address. This allows services to move from one location to another without affecting future consumers (current consumers may be informed of new location).

2.4 Web-service based approaches

Different implementations are available to support the interactions between service consumers and service providers in an SOA. The implementation alternatives impact important quality attributes of the system, such as interoperability and modifiability. In a pure Web services solution, the "Simple Object Access protocol" (SOAP) is used. This is has been the common approach for several years. However, the Representational State Transfer (REST) has been proposed as a simpler approach and is currently becoming quite popular, namely in the Future Internet domain. This section describes these two alternatives.

2.4.1 SOAP-based Web services

SOAP-based Web services technology has been used to implement SOAs. It is a protocol specification for exchanging structured information between two Web services and relies on several elements:

- XML Information Set – W3C specification describing an abstract data model of an XML document in terms of a set of information items. This Information Set is used to describe the message format of requests and responses in a service interaction.
- Hypertext Transfer Protocol (HTTP) – transport of the request and response messages is made by the HTTP protocol (as an alternative, the Simple Mail Transfer Protocol (SMTP) may also be used).
- Web Services Description/Definition Language (WSDL) – XML-based language used in the description of the Web services interface.

Basically, communication between the consumer and provider of a given service is accomplished thought the SOAP protocol, while its interface is defined by a WSDL description. Two types of communication are commonly implemented: RPC-encoded and document-literal.

RPC-Encoded SOAP

In the RPC style, the SOAP message is similar to an XML-based remote procedure call. The name and type of each argument is specified in service's WSDL definition. Accordingly, the body of the SOAP request specifies the name of the procedure being invoked and all of their arguments (names and values). An "encoded" attribute in the WSDL definition indicates that data is serialised using a standard encoding format (SOAP specification specifies rules for how to encode primitive data types, strings, and arrays).

Document-Literal SOAP

In the Document-Literal SOAP, the message body can contain arbitrary XML elements, they don't follow a standard structure as in the RPC style but are formatted and interpreted using the rules specified in XML schemas created by the service developer (this is indicated in the WSDL's "literal" attribute). Thus, XML schemas (XSD) define the data structure of requests and responses.

Even though the RPC-Encoded style may be simpler to use, it is not recommended as interoperability problems can arise from the deficiencies in the SOAP-encoding specifications (Bianco et al., 2007). The Document-Literal is the current recommended style by the WS-I organization (www.ws-i.org) as is more interoperable.

2.4.2 REST-based Web services

This approach is more aligned with the Internet. It is based on the concept of "resource", which is seen as a piece of information uniquely identified by a URI (uniform resource identifier) in a way which is similar to the links used in web browsers.

A REST–based invocation of a service (i.e., the "resource" identified by a URI) relies on the HTTP protocol via the following basic operations: POST, GET, PUT, and DELETE. In a REST design, the invocation of each such basic operation on a resource URI corresponds to the operations commonly used in information systems to create, retrieve, update, and remove (CRUD) a "resource". For example, if the service consumer sends a HTTP POST request on the "resource" URI http://www.weather.com/current/zip/pt-4200-395, it is asking the service provider to create the data "representing" the current weather in zip code pt-4200-395 by using the data passed along with the HTTP request. Accordingly, a GET request would request the service provider to return the data representing the current weather for pt-4200-395. A PUT request for the same "resource" requests the service provider to represent the weather for the given zip code with new data sent along the request. Finally, the DELETE request towards that same zip code would request the service provider to delete the representation of the identified "resource" URI.

As described, REST-based Web services have a uniform interface: the service is exposed as an information resource which is the target of the four basic CRUD operations unlike the operations used in the SOAP-based style where any type of operation can be defined. Moreover, the "representation of resources" has to be defined (current approaches are based on XML and JSON). By its definition, a REST service is stateless (state of conversation across several service interactions is not kept by the REST service).

3 ROS communication model

The Robot Operating System (ROS) is a set of software libraries and tools (framework) that help developers to build robot applications (www.ros.org). The framework provides standard operating system facilities such as hardware abstraction, low-level device control, implementation of commonly used functionalities, message passing between processes, and package management. Runtime processing takes place on ROS nodes, a cohesive set of computer instructions aiming to implement a given functionality in a robot or sets of robots. Usually, a ROS-based system is a set of ROS nodes that communicate through message exchange. Accordingly, a message is the basic communication element containing data send by one node to another (or several) node. The framework defines a set of messages but developers are free to construct new message types.

A ROS node may make available in the network any number of services. Interaction between services obeys to the request-response communication model. Invocation of a service is implemented by a request message and the synchronous response by another message. Type of the messages (its internal data structure) must be defined at design time.

A one-to-many communication model is available via publish subscribe communication model. A topic is published by a given node allowing interested nodes in the ROS system to subscribe to that topic. Each topic is related to a message type so that instances of this message type are created by the publisher and all subscribed ROS nodes receive that message without having to request it. Communication is thus one way, from a publisher to subscribers, without any guarantee of reception.

A special ROS node in the network supports service-based and publish-subscribe communication between nodes. It is the ROS Master, responsible for knowing which service providers and publishers/subscribers are active in the network. The ROS node provides a service whereby:

- Nodes acting as service providers can register the services they provide as well as change that registration in the following future;
- Nodes acting as service consumers can retrieve the identification/location of the node that provides a service having a specific name, as well as be notified of a change in the service registration;
- Nodes acting as topic publishers can request the publication of a given topic;
- Nodes acting as topic subscriber can request their interest in the topic.

4 STAMINA service-oriented architecture for robot fleets

4.1 Abstract architecture

Figure 1 represents STAMINA system at a very abstract level. At the bottom level, each STAMINA robot, apart from all the hardware components, is managed by two ICT elements: a Task Planner and a several Skill Manager (see D3.1 "Preliminary report on skill architecture"). These two elements (and the skill-oriented ones not represented in the picture) will operate on top of the robot's operating system (ROS), running as ROS Nodes, in the most autonomous way possible. At a top level, Logistic Planner and Mission Planner will have the general visibly over the robot fleet, and as such, their operating environment will not be ROS but Windows or Linux.

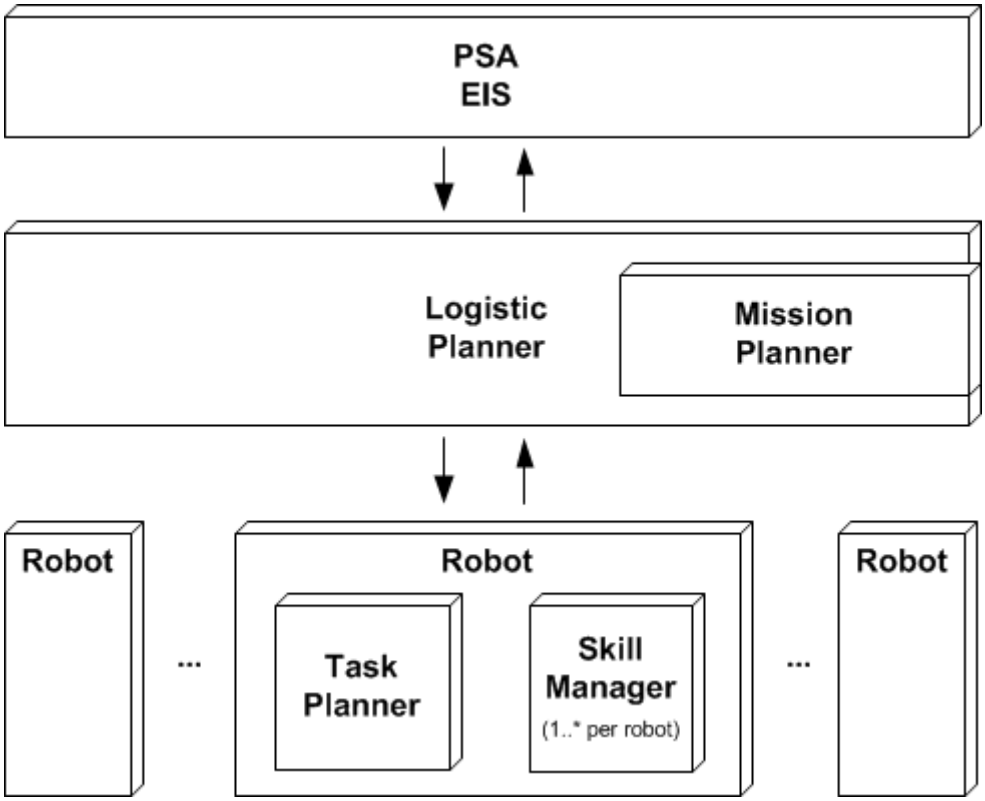

Figure 1 – STAMINA abstract architecture.

4.2 Abstract service-oriented architecture

Figure 2 refines the previous abstract architecture by identifying the high level elements in STAMINA system and their interactions (an arrow between two STAMINA elements represents the consumption of a service by a service consumer). This diagram constitutes the present abstract service-oriented architecture according to current knowledge and consensus in the project. A description of these interactions follows, followed by a table describing each such service interaction. In the near future, this specification will be further detailed so as to describe the data elements of each service interaction and their service interface in terms of technological elements (e.g. WSDL, ROS services, topics and messages, etc.).

The PSA EIS elements comprise several elements that were previously described in Deliverable 4.2.1. These elements provide information about existing part references, their packaging structure, each kit structure, the structure of the furniture positioned in the kitting zone as well as the implantation structure of mobile objects (empty kits, full kits, …). This information is to be maintained by the Logistic Planner, together with the most dynamic information that represents a kitting order in STAMINA system. As described in Deliverable D4.2.1, the FAC object[1] is created by PSA EIS 15 to 20 minutes before the correspondent vehicle enters the assembly line (a FAC is related to only one vehicle). This pool of FACs is maintained and transmitted to Logistic Planner in a pro-active manner. Additionally, a signal identifying the entry of a vehicle in the assembly line is also delivered to the Logistic Planner. A counter, decremented on each signal and incremented on each built kit, is maintained by the Logistic Planner that, combined with a threshold level (defines the number of kits that may be placed in the kitting zone's output conveyor), will control the number of FAC objects that are processed.

Periodically (and depending on the actual value of the previous counter) FAC objects are sent by the Logistic Planner to the Mission Planner, asking it to define missions and their assignments to robots in the fleet. In order to support this, Mission Planner requests from the Logistic Planner all needed information in order to identify the current robot status in the fleet, their abstract skills, constraints and maps about the kitting zone. A preliminary overview of the Mission Planner is provided by (STAMINA, 2014d). According to this, two versions of the mission planner are foreseen: (i) a Test-Case version, a functional but limited version respecting the constraints of the current factory environment, robot systems, and needs of the project test sprints (e.g., single robot operation), and (ii) a Blue-Sky version, a comprehensive version for controlling robot fleets that is not constrained by the limitations of the present factory environment and existing robots.

As soon as the Mission Planner specifies the missions for the requested FAC objects, Logistic Planner is requested to transfer the information about the FACs and missions to the selected robots in the fleet. This involves interacting with each Task Planner running in each selected robot and transferring that information.

According to the planning in the mission, the assigned Task Planner starts interacting with the local (at the robot) Skill Managers, in order to know their exact skills and match them with the abstract skills initially defined, plan their sequence and prepare the input parameters for its execution. Subsequently, each Skill Manager is requested to start executing a skill informing the Task Planner about the main execution phases. A preliminary description of the skills framework is provided by (STAMINA, 2014c).

[1] Presently, in order to have a common understood language inside STAMINA team, the term FAC is the preferred term to reflect a kitting order inside STAMINA. In the future, the term "kitting order" may be a preferred term.

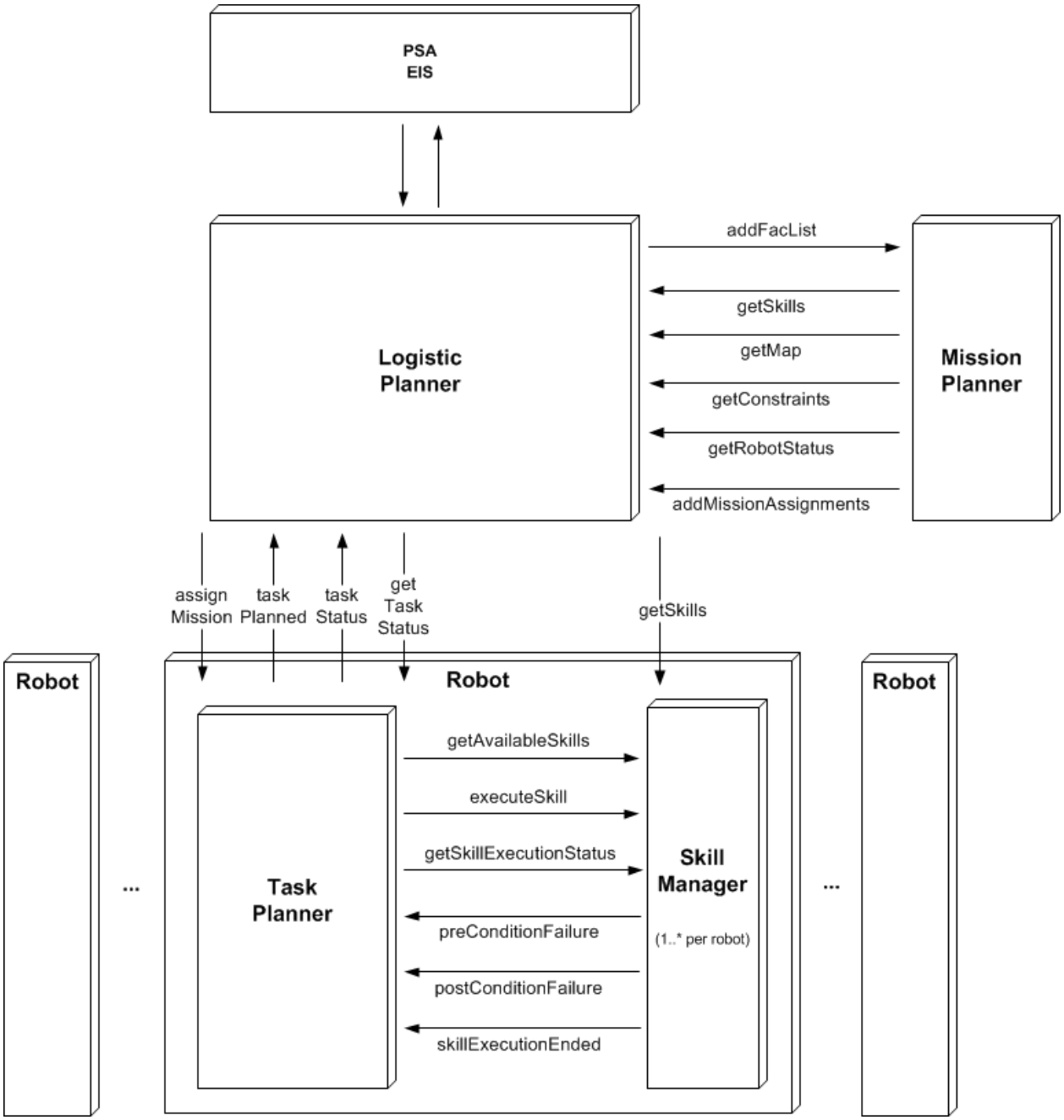

Figure 2 – STAMINA abstract service-oriented architecture.

As identified in the previous chapter, a service registry is required to help service consumers and providers to establish connections and start communicating. A Service Registry is proposed for STAMINA[2] (see Figure 3) so that:

- service providers may register and unregister the service in a dynamic way (at least when the start and stop working)
- and service consumers know the exact description of a required service (service description and service location in the network) and be informed about changes in their registration.

[2] The ROS communication model implements a similar registry, named ROS Master, for all ROS nodes. STAMINA service registry will need to interact with the ROS Master (details to be worked out in the near time).

A keepAlive message is defined enabling STAMINA components to inform the Service Registry about its status in the network.

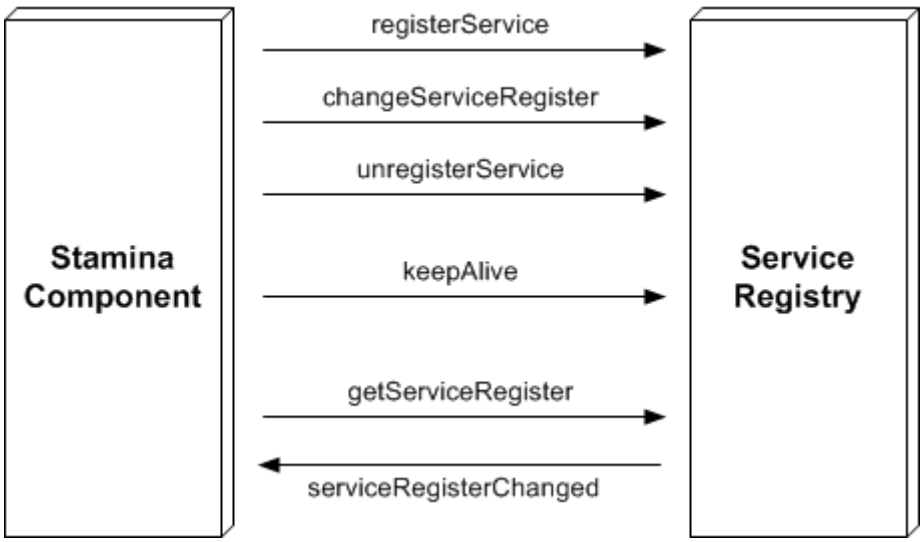

Figure 3 – STAMINA Service Registry.

Table 1 identifies the STAMINA services defined so far for the abstract service-oriented architecture presented above. For each service its name and purpose are briefly identified, followed by details about its usage. The technical computer artefacts (including data models) will be subsequently built after this specification.

Table 1 – STAMINA services API

Service	Purpose	Details
addFacList	Logistic Planner requests the Mission Planning for the planning of the Mission of a set of FAC objects. A Mission targets the construction of one or two kits (in case robots have this skill).	Service invoker: Logistic Planner. Service provider: Mission Planner. Type: request-response. Input parameters: array with FAC objects. Output parameters: void.
getSkills	Mission Planner asks the Logistics Planner for the current skills specification	Service invoker: Mission Planner. Service provider: Logistic Planner. Type: request-response. Input parameters: void. Output parameters: array with skill specifications for each robot.
getMap[3]	Mission Planner asks the Logistics Planner for the current kitting zone map.	Service invoker: Mission Planner. Service provider: Logistic Planner. Type: request-response.

[3] Blue-Sky version.

		Input parameters: void. Output parameters: encoding of the Kitting zone map for use with the Mission Planner.
getConstraints[3]	Mission Planner asks the Logistics Planner for the current constraints on assigning missions.	Service invoker: Mission Planner. Service provider: Logistic Planner. Type: request-response. Input parameters: void. Output parameters: encoding of any external constraints on the possible mission assignments.
getRobotStatus[3]	Mission Planner asks the Logistics Planner for the current robot status information.	Service invoker: Mission Planner. Service provider: Logistic Planner. Type: request-response. Input parameters: void. Output parameters: Encoding of all robot status information stored by the Logistics Planner. This may include elements such as battery level and robot availability that could affect mission planning.
addMissionAssignments	Mission Planner request Logistics Planner to add mission assignments to be sent to robot fleet. A mission assignment contains the FAC, the chosen abstract skills, and the selected robot.	Service invoker: Mission Planner. Service provider: Logistic Planner. Type: request-response. Input parameters: Array with mission assignments. Assigns FACs to members of the robot fleet. Output parameters: void.
assignMission	Logistic Planner assigns mission to the appropriate Task Planner (at a robot in the robot fleet).	Service invoker: Logistic Planner. Service provider: Task Planner. Type: request-response. Input parameters: Mission assignment Output parameters: void.
taskPlanned	Task Planner informs the Logistic Planner that a task was planned.	Service invoker: Task Planner. Service provider: Logistic Planner. Type: one-way. Input parameters: identification of mission/task and the sequence of skills to be executed.
taskStatus	Task Planner informs the Logistic Planner about the last change of state in the execution of the Task.	Service invoker: Task Planner. Service provider: Logistic Planner. Type: request-response. Input parameters: identification of

		mission/task/skill, and state change (taskStarted, taskCancelled, taskFinished, ...).
getTaskStatus	Logistic Planner requests the Task Planner to state the current state of execution of task.	Service invoker: Logistic Planner. Service provider: Task Planner. Type: request-response. Input parameters: identification of mission/task. Output parameters: identification of mission/task, and current execution status of each skill.
getSkills	Logistic Planner requests a given Skill Manager to identify all implemented skills. A robot may have several Skill Managers.	Service invoker: Logistic Planner. Service provider: Skill Manager. Type: request-response. Input parameters: void. Output parameters: array containing the description of each implemented skill.
getAvailableSkills	Task Planner requests a given Skill Manager in the same robot to identify its implemented skills.	Service invoker: Task Planner. Service provider: Skill Manager. Type: request-response. Input parameters: void. Output parameters: array containing the description of each implemented skill.
executeSkill	Task Planner requests a given Skill Manager to start executing a given skill.	Service invoker: Task Planner. Service provider: Skill Manager. Type: request-response. Input parameters: identification of skill and input parameters. Output parameters:void.
getSkillExecutionStatus	Task Planner requests a given Skill Manager about the current status of execution of a started skill.	Service invoker: Task Planner. Service provider: Skill Manager. Type: request-response. Input parameters: identification of skill. Output parameters: current status of execution of skill (skill started, precondition evaluation, executing body, post condition evaluation).
preConditionFailure	Skill Manager informs Task Planner that a given task stopped execution because of precondition failure.	Service invoker: Skill Manager. Service provider: Task Planner. Type: one-way. Input parameters: identification of skill

		and failure.
postConditionFailure	Skill Manager informs Task Planner that a given task stopped execution because of post condition failure.	Service invoker: Skill Manager. Service provider: Task Planner. Type: one-way. Input parameters: identification of skill and failure.
skillExecutionEnded	Skill Manager informs Task Planner that a given task ended successfully its execution.	Service invoker: Skill Manager. Service provider: Task Planner. Type: one-way. Input parameters: identification of skill.