Chapter 6

System integration

Endowing a humanoid torso with autonomous manipulation abilities involves different research topics and requires the integration of many components. All the different building blocks of the system identified in Chapter 2 and implemented in Chapters 3, 4 and 5 need to work together in a coordinated fashion. Hence, it is necessary to provide an integration architecture that allows the modules to communicate to each other. It is also important that the proposed solution supports the paradigm of manipulation primitives and task descriptions introduced in Chapter 3. Finally, it has to be general, such that the integration and implementation of other research projects is possible.

A general approach to integrate different components into a system is the use of middlewares (e.g. ROS, Yet Another Robot Platform (YARP)). However, using a well known middleware and implementing the system components as modules, does not satisfy all the requirements. In big projects with many modules like the one presented in this thesis, structureless implementation and heterogeneous inputs and outputs results in chaos. Therefore, to ease the code maintenance and the integration of new modules, an integration architecture that defines a common structure, concepts, coding style and documentation is required.

In this chapter we describe the software architecture developed to integrate all the components of the contact-based manipulation system, it consists of a modular four layered architecture with three different data flows. The different layers and data flows are designed to support natively the manipulation primitives paradigm and the definition of tasks as introduced in Chapter 3.

Moreover, there are several built-in abilities (e.g. inverse kinematics, object detection) that are supposed to be available by the manipulation primitives, contact perception or contact prediction. Those abilities are also presented in this chapter. Finally, the portability of the architecture is addressed and its implementation on another robotic platform is shown.

6.1 Software architecture

To control all the modules within an integrated platform, we have implemented a layered system that enables us to interact with the robot at different abstraction levels (see Fig. 6.1). The core of the architecture is formed by three layers: service, primitive and task. These core layers communicate with the simulator and the robot hardware through the interface layer. There are other secondary components such as GUI, robot models and databases. Each layer is composed by modules that run in parallel and communicate with each other using three main types of messages. The *data messages* are the input/output of the modules. They contain any type of information, raw or processed (e.g. joint status, camera image, object positions). The *control messages* change the parameters (e.g. threshold, loop rate) and the status (e.g. run, stop) of the modules, all the modules accept by default the commands run, stop or reset. Finally, the *event messages* contain information about a detected situation (e.g. an object is localized or grasped successfully, motion detected).

ROS is used to handle the messages between the different system parts. ROS is a middleware that provides a message passing framework among other features. Typically, a ROS-based system is composed of multiple ROS nodes. A ROS node is a process that can send and receive data using two different methods: *ROS topics* for asynchronous n-to-n data streams and *ROS services* that provide a request/response 1-to-1 mechanism.

Each module belongs to one of the four layers depending on its purpose and its implementation. Hardware interfaces and drivers belong to the *interface* layer. Above them, modules that perform basic operations such as sensor processing or motor control, belong to the *service* layer. Mid-level modules like manipulation primitives (grasp, push, etc.) or perceptual primitives (locate object, wait for event, etc.) are classified into the *primitive* layer. Primitives can rely on services to accomplish their goals. More complex modules that require several primitives, services or other tasks to work, are classified into the *task* layer (e.g. pick and place). Depending on the layer where a module belongs to, it must follow specific implementation rules.

6.1.1 Robot and simulator interfaces

The robot interface layer converts the messages back and forth from hardware drivers to the ROS messages used by our system. Exactly as for the real hardware interface, we have implemented interfaces for the OpenRAVE/OpenGRASP [León et al., 2010] simulator and Gazebo [Koenig and Howard, 2004]. The simulator interfaces provide the same inputs and outputs than the hardware interfaces. This allows the upper levels to work without knowing whether they are controlling the real or the simulated robot. As detailed in Chapter 5, the simulator is also used as a prediction engine.

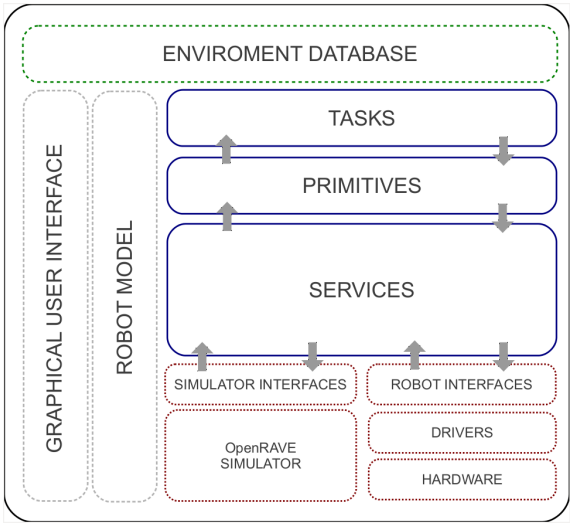


Figure 6.1: Integration software diagram.

6.1.2 Services

The *services* are basic processes that receive an input and provide an output. Thus, services are mostly used to obtain high level information from raw sensor data and to send control data to the interface layer. The purpose of the service layer is to provide the building blocks for higher layers. Services are characterized by the following properties: i) they are continuous non blocking loops that never stop by themselves, ii) they generate at least one output and require zero or more inputs, but they do not generate control messages.

6.1.3 Primitives

The *primitives* define higher level actions or processes. A primitive is a control loop that gets processed data from services, generates events and sends commands to the underlying services. They are defined by the following properties: i) They are continuous non blocking loops that never stop by themselves. ii) They generate at least one output and can have but do not require inputs. They also generate events towards the task layer. The role of the primitives is to use and manage services to control the robot (grasp, move, transport or look at), perceive (recognize or localize objects) and detect special situations (motion detected, object lost).

6.1.4 Tasks

Tasks represent the highest level of our architecture and use primitives as building blocks to generate desired behaviours. A task (as defined in Chapter 3) can be described as a cyclic, directed, connected and labelled multi-graph, where the nodes are primitives

and the arcs are events. An example of a task description that performs active object tracking with head movements is depicted in Fig. 6.2. It is used in Fig. 6.3 together with a manipulation task to compose a task that grabs an object while looking at it. Tasks have the following properties: i) They can end, do not need to be continuous. ii) They can have multiple inputs and outputs. iii) They generate events.

The role of a task is to coordinate the execution of the primitives, by generating control messages and changing the data flow between primitives and services. Multiple tasks can run in parallel and can also be coordinated by other tasks.

Command message Messages sent to the modules to change the configuration parameters or change their execution status. The default commands for each module are: run, stop and reset.

Data message Generic message used to send data from one module to another module.

Event message Message that is generated by the detection of a specific condition or event (e.g contact detected, object lost, object detected).

Module generic component of the system. Is composed by a ROS node with defined inputs and outputs (ROS topics and services). Its constraints and default functionalities depend on the layer it belongs to.

Service basic processes that receive an input and provide an output. Are continuous non blocking processes that never stop by themselves. Generate at least one output and require zero or more inputs. Do not generate control messages. (e.g. filters or joint controllers)

Primitive define higher level actions or processes. A primitive is a control loop that gets processed data from services, generates events and sends commands to the service layer (e.g. grasp, transport).

Task represent the highest level modules. They use primitives as building blocks to generate the desired behaviours. A task (as defined in Chapter 3) can be described as a cyclic, directed, connected and labelled multi-graph, where the nodes are primitives and the arcs are events (e.g. clear the table).

6.2 Implementation and task setup

6.2.1 Module implementation tool

The software architecture does not only settle the concepts but also provides a tool to automatically generate code. It follows the coding style and guides the developer through the process of creating a new service, primitive or task. A module can be described by its inputs, outputs, parameters and events. For the creation of a new

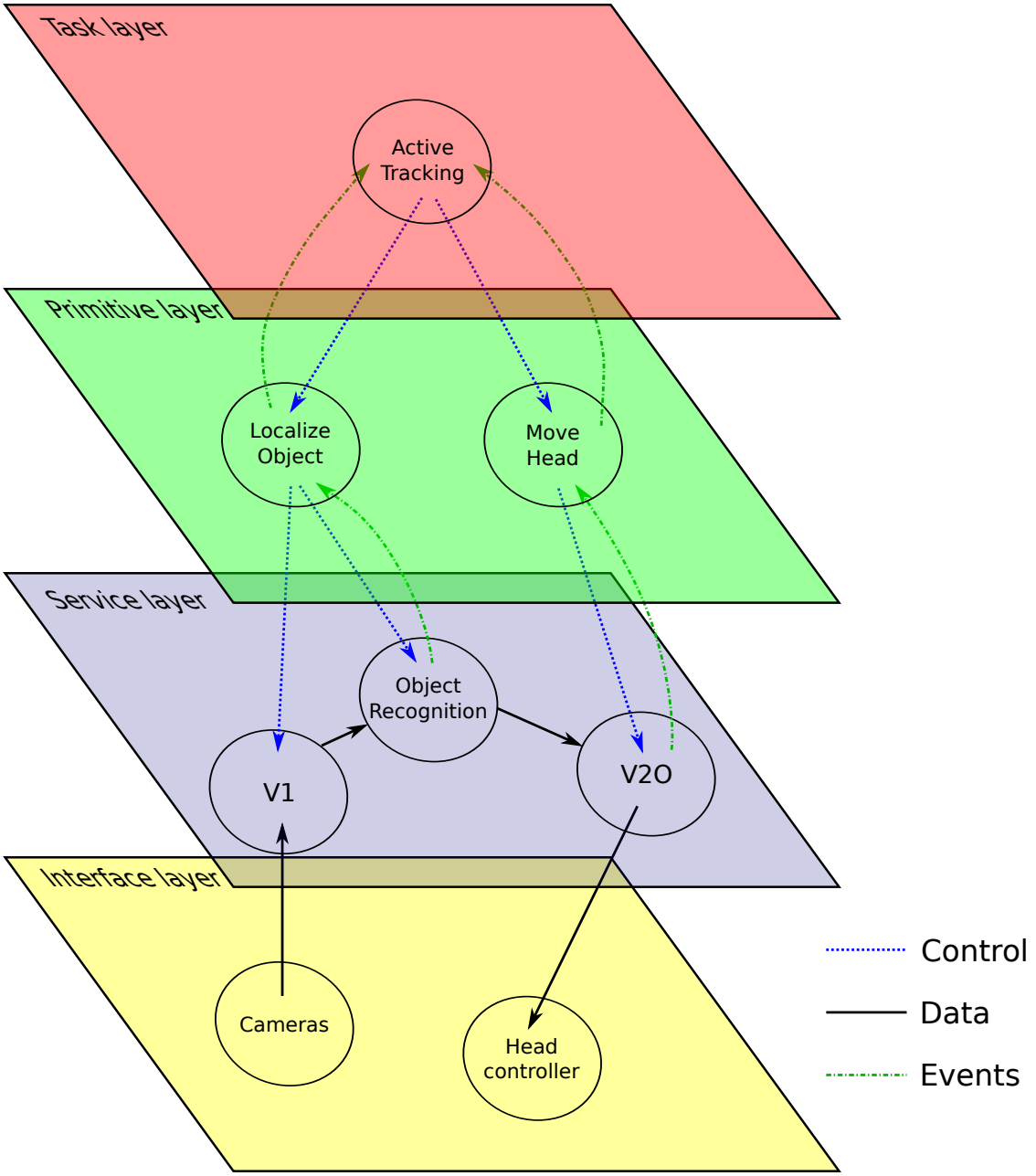


Figure 6.2: Active tracking task. It consists of two primitives, *Localize Object* and *Move Head*. The former employs two services to extract visual features and localize the target object. The latter contains a service that calculates the eye movement required to gaze at the target. Both primitives trigger an event when their state changes.

system module (service, primitive or task), we provide an interpreter that uses the XML description of the module and creates doxygen friendly code stubs with all the communication and parameter handling. It also takes care of configuration files, cmake files and launchfiles. Advantages of the interpreter are: 1) the developers do not have to waste time creating the code stubs, configuration files and cmake files, which is a mechanical task that can be automated. 2) All the modules have a similar structure making the code standard and easily understandable by other developers, the XML description is human readable and provides detailed module information at a glance. 3) Automatic code generation eliminates the possible errors introduced in those parts of the code.

6.2.2 Configuration and parameter setting

Each module XML description contains the parameters, inputs and outputs of the module. The configuration (inputs, outputs and parameters) of each module can be statically set in a YAML Ain't Markup Language (YAML) configuration file. Moreover, the module configuration can also be modified online as the task is being executed using command messages. To configure a module to use it in an experiment, it is mandatory to define the connections of its inputs and outputs. Although the modules provide default values for their parameters, it is important to set up them accordingly to the task.

6.2.3 Task example

A simple task devoted to recognize and actively track an object is depicted in Fig. 6.2. Assuming that all the modules required are implemented, the configuration files for each module should be provided. The configuration files contain, for each module, the parameter values and the input and output connections. For example, the YAML file for the *Object Recognition* service contains the input connected to the *V1* service, the output connected to the *V2O* service and the parameter with the object id. The primitives and tasks are configured using the same process.

More complex experiments using this architecture have been presented in Chapter 3 where the robot is able to empty a box full of objects and grasp a bottle with one hand and unscrew its cap with the other hand. Another experiment using a different robotic platform is detailed in Sec. 6.5 where the task implemented to solve the Amazon Picking Challenge is presented.

6.3 Implemented modules

A set of services can be connected in series, connecting the output to the next service input. This kind of configuration, where the raw input is processed by several services, is called pipeline. Pipelines are used to implement complex processes as a series of

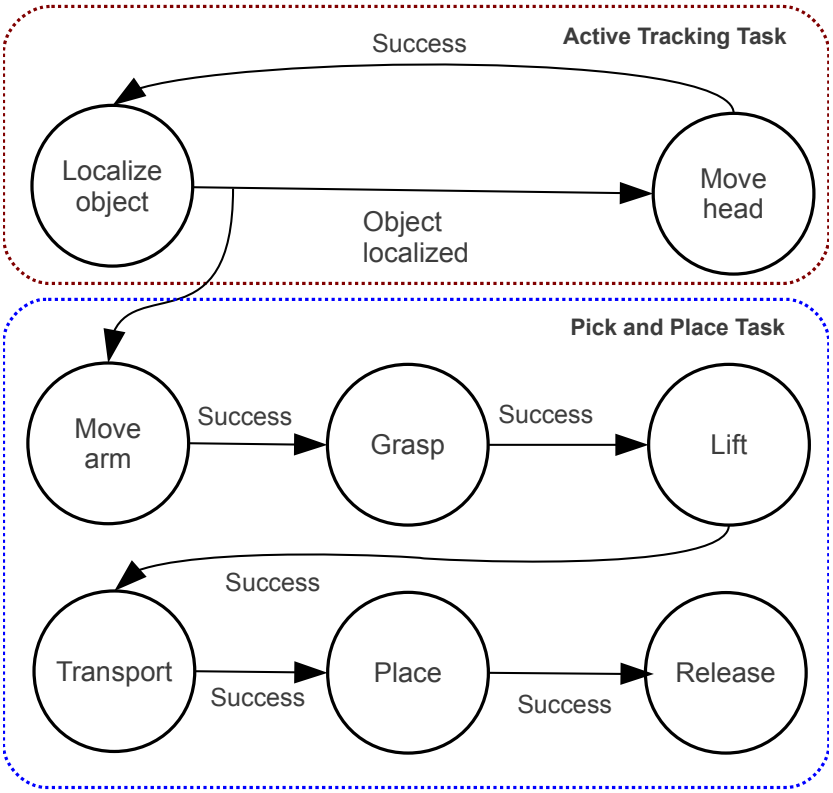


Figure 6.3: Cooperation among tasks. The robot executes a task which consists of grasping and moving the target object (*pick and place*), that requires six primitives. Meanwhile, the robot actively tracks the moved object.

simple operations. An example of a pipeline is detailed in this section where the visual processing pipeline and the grasp planning pipeline are outlined.

This section describes the basic modules and pipelines implemented on the robot either as services or primitives. They are used together with other primitives and services, to create new tasks.

6.3.1 Joint controller services

The low level control is performed by three services that interface the architecture with the robot or simulator interfaces. Since each joint can receive different control signals concurrently from different controllers, the *joint state subscriber continuous* combines the control inputs, using a configurable weight, and sends the result to the interfaces at a fixed frequency of 200Hz. *Joint state publisher continuous* performs the inverse task: it gathers the information from different hardware interfaces, combines them and provides the information of all the joints of the robot in a single message at 200Hz. Using the robot model and the joint values, the ROS tool *robot state publisher* provides the transformation tree for the entire robot using the TF library[1]. TF is a ROS library that provides tools for geometric transformations of different data types among different reference frames. The TF tree is the representation of the reference frames available on the system, the tree for Tombatossals is depicted in Fig. 6.4.

6.3.2 Arm controllers and planning

The arm control is composed of several services grouped into two pipelines. The inverse kinematics pipeline converts the end effector target pose, velocity or wrench specified in the Cartesian space (w.r.t. any known frame), to robot joint velocities. The planning pipeline converts the target pose into a plan in joint space and executes it.

The *inverse kinematics pipeline* is composed of two services. First, the *Cartesian controller* is in charge of converting any target input (position, velocity or wrench) into a Cartesian velocity command w.r.t. robot_base using the TF library (see Fig. 6.4). Second, the *ik solver* transforms a Cartesian velocity command w.r.t. robot_base into joint velocities. It is an inverse kinematics iterative solver based on the pseudo inverse of the Jacobian matrix. The solver is provided by the KDL library [Smits, 2015] and requires the robot model, the kinematic chain and the controlled frame.

The *planning pipeline* is a set of services that interface with the MoveIt! [Sucan and Chitta, 2015] framework. The first service of the pipeline is the *ros moveit interface* that converts the input target pose into a MoveIt! message and forwards it to the planning pipeline. The planning request is then received by the *move group* service, the core service of the pipeline. This service uses the robot description, the environment description and the depth sensor input to obtain a collision free plan from the current

[1]See `http://wiki.ros.org/robot_state_publisher`

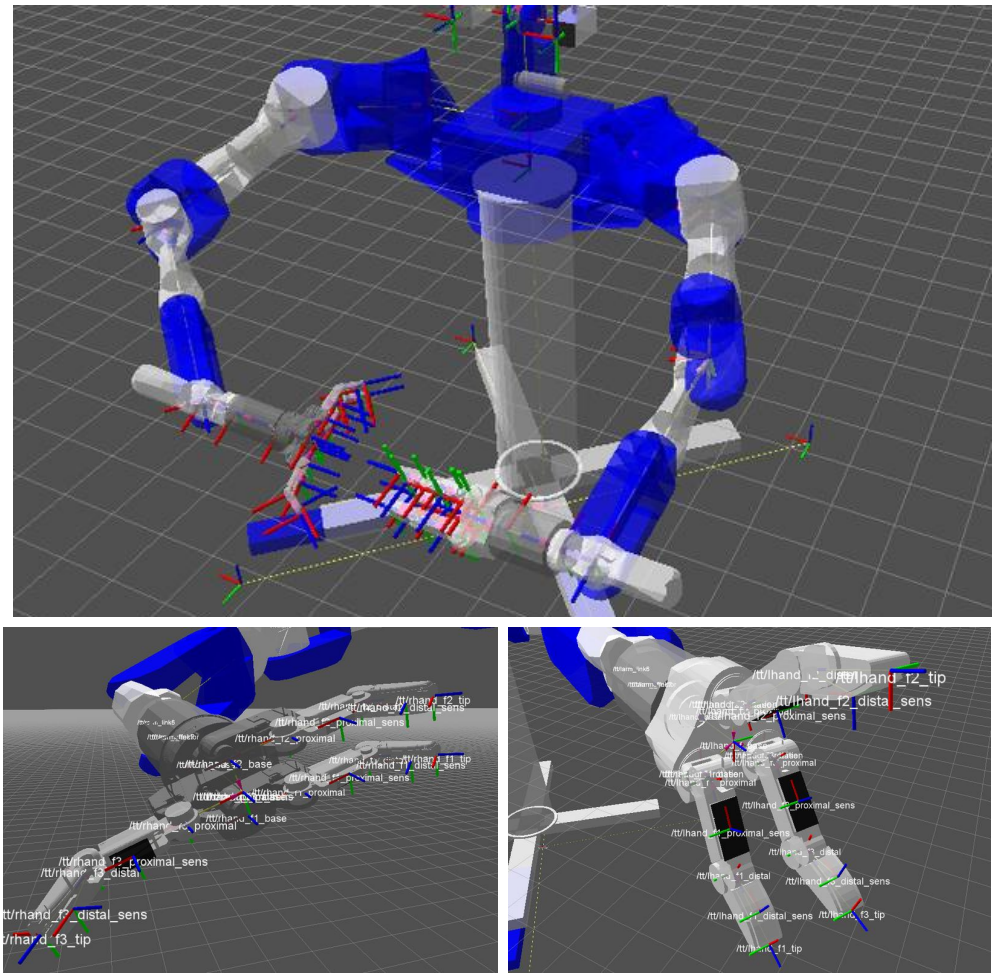


Figure 6.4: Visualization of the TF tree for Tombatossals. Top: full robot. Bottom left: right hand. Bottom right: left hand.

arm pose to the target pose. To obtain the plan, the *move group* service uses one of the algorithms available in Open Motion Planning Library (OMPL). The algorithm is selected and configured through the *move group* service commands. The result of the planning process is a trajectory composed of a waypoint list. Each waypoint contains a target joint configuration of the robot. Finally, the planned trajectory is received and executed by the *robot trajectory* service that moves the robot joints from waypoint to waypoint. While moving, the force and RGBD sensors are monitored to watch out for unexpected collisions. If a collision is detected the execution is aborted.

The target inputs can be specified on any known reference frame connected to the robot TF tree. This allows the upper layers to set up a task frame and specify their commands in the task frame without the need of performing conversions to the robot frame. Using a task frame is very convenient to specify positions, velocities and forces. It is easier to specify the rotation of a door knob in the door knob frame than specify the same rotation on an arbitrary frame not aligned with the door knob rotation axis.

6.3.3 Visual perception pipelines

The visual system of the robot consists of a Kinect sensor and two cameras (more details about the visual sensors are provided in Appendix A.1). Several services that exploit common libraries such as PCL [Rusu and Cousins, 2011] or Opencv [Bradski and Kaehler, 2008] have been implemented to process the visual information.

3D pipeline

The 3D visual pipeline (Fig. 6.5), processes the cue from the RGBD camera and provides a segmented point cloud for each object in the Region Of Interest (ROI). The first service of this pipeline is the *plane detector*. This service detects the principal plane of the scene using RANSAC. Based on that plane, the ROI is calculated extruding the bounding box of the points that belong to the principal plane. As the principal plane is not expected to change fast, this service runs at 1Hz to save computational power. The *ROI filter service* receives two inputs: a ROI (defined by 8 vertices) and a point cloud. Then it filters out the points outside the ROI. Then the *self filter service* is applied, this module uses a spherical model of the robot in conjunction with the current robot joint positions to remove the points in the point cloud that are part of the robot (the spherical model of the robot is detailed in Appendix B). After this process, the point cloud only has points that belong to objects in the workspace. Finally, the remaining points are processed by the *cluster extractor service*, which performs an euclidean clustering on the input point cloud and publishes serially each cluster in a separated point cloud. On our current setup, the pipeline is able to provide the extracted clusters at 30Hz. However, the computational complexity of the clustering algorithm depends on the number of points to be processed, we consistently obtained a 30Hz rate but that performance might decay for larger objects with more points.

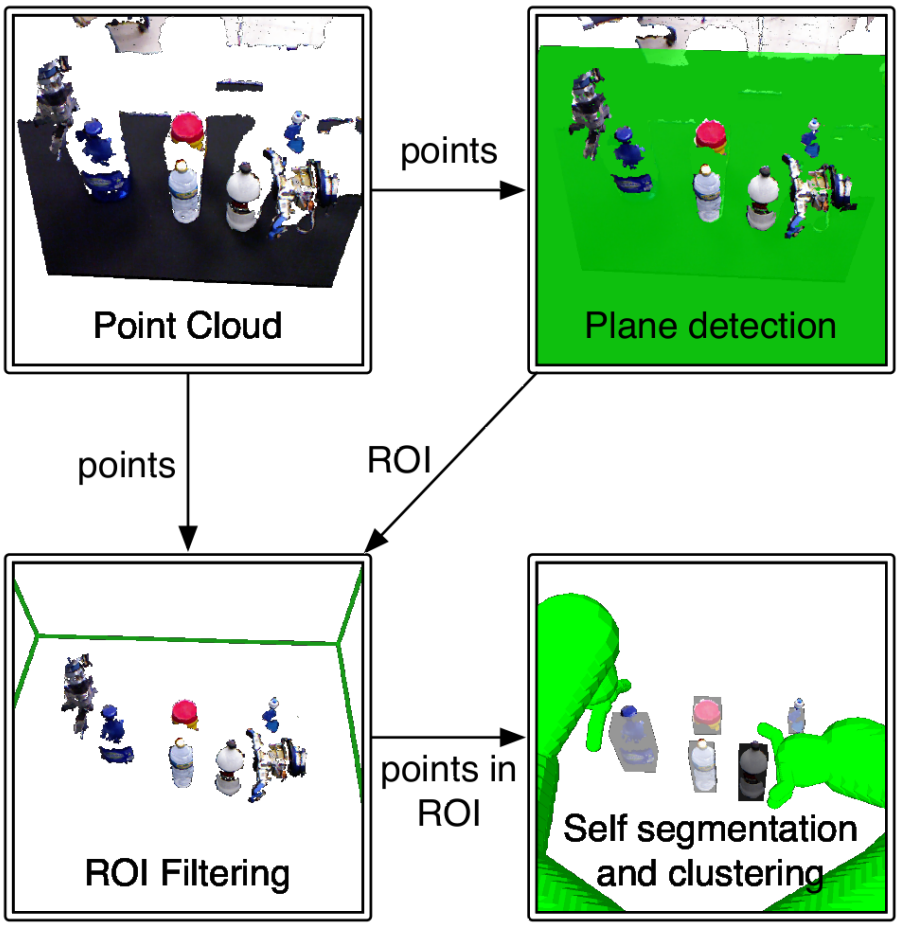


Figure 6.5: The 3D vision pipeline for object segmentation. The point cloud is processed by the plane detector and the principal plane is obtained. The region of interest (ROI) is calculated extruding the bounding box of the points that belong to the principal plane and sent to the ROI filter service. The ROI filter service processes the input point cloud and applies the ROI obtained from the plane detector. From the resulting points the self-filter service removes the points that belong to the robot, then a clustering algorithm detects the objects.

All the features in this pipeline are implemented using the algorithms and filters available in the PCL library [Rusu and Cousins, 2011]. The pose of the RGBD camera is known in advance. To calibrate it, an AR marker was attached to a known frame of the robot and calculated the camera-robot transform. This calibration allows us to add the RGBD camera frame to the TF tree and the visual system can output the data w.r.t. robot base if necessary.

6.3.4 Contact perception pipeline

There are several sensors able to detect physical interaction with the environment. Although each sensor can be accessed independently from either services or primitives, the contact perception services implement a sensor fusion approach combining all the available sensors to produce contact information which can be used by other services or primitives. The *contact_hypothesis_fuser* service provides the output of the combination of all the contact hypotheses generated by the other services of the contact perception pipeline. The details of the fusion process and contact hypothesis generation from tactile, force, vision, simulation and context are already provided in Chapter 4.

6.3.5 Grasp synthesis pipeline

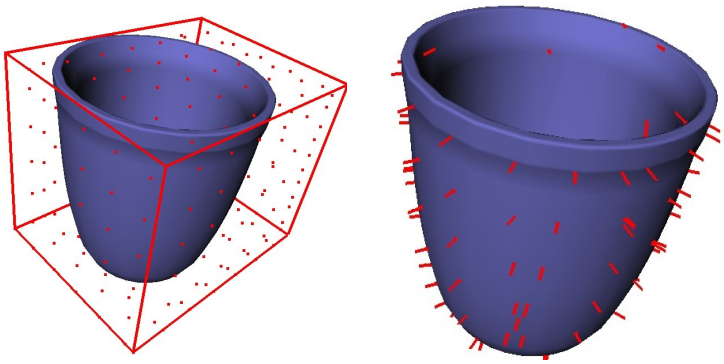
In this thesis we use sensor-based reactive approaches for the implementation of manipulation skills, thus the grasp planning algorithms used do not provide exact contact points but Approach Vectors (AVs). AVs determine the starting position for object manipulation strategies. There are two implemented methods to generate AVs: the *simple grasp generator* based on the perceived object geometry and the *openrave grasp generator* based on the object model.

The *simple grasp generator* implements a method inspired by [Rombokas et al., 2012], that performs PCA on the object's segmented point cloud to generate approach vectors that are parallel to its principal axes. The position is determined by the intersection of each principal axis and the bounding box of the perceived object geometry.

The *openrave grasp generator* uses the object model (6D registered or reconstructed) and the robot model in the OpenRAVE grasp generator to produce the AVs. First, the surface of the object's bounding box is uniformly sampled. Second, the intersection of the object and a ray originating from each sampled point going inward is taken. Finally, the normal of the object's surface from each of these intersection points is taken to be the approaching direction of the end-effector, see Fig. 6.6. More details about the OpenRAVE AV generator can be found in the OpenRAVE online documentation[2].

To select a single approach vector or reduce the search space for the final AVs selection, the generated AVs can be filtered using different services. The *reachability filter service* filters out the AVs that are not reachable by the selected robot arm. The *Collision-free*

[2] `http://openrave.org/docs/0.8.2/openravepy/databases.grasping/`



(a) The surface of the bounding box of the object is uniformly sampled.

(b) The intersection of the object and a ray originating from each sampled point going inward is taken.

Figure 6.6: Example of the approach vectors generated by the OpenRAVE grasp planning plugin. Images obtained from the OpenRAVE online documentation.

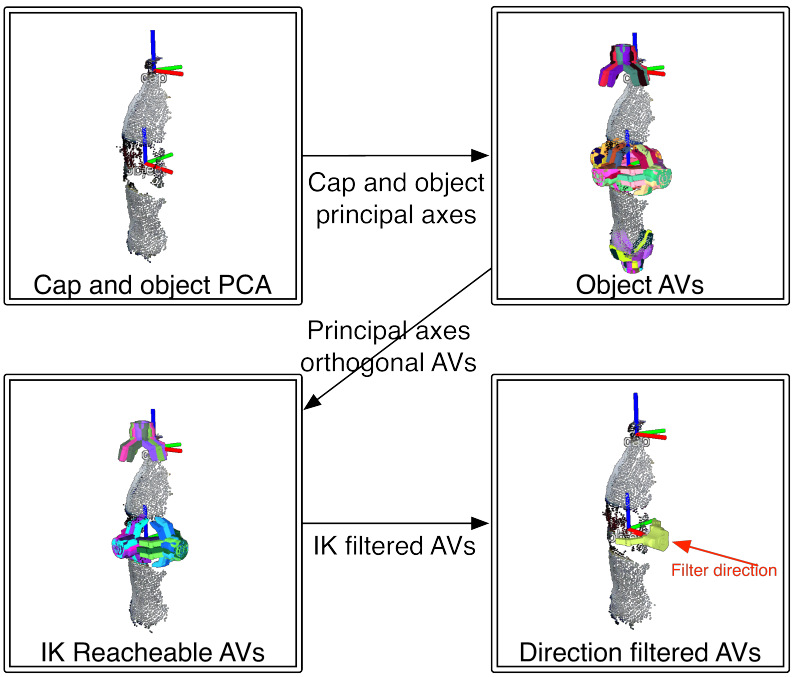


Figure 6.7: Example of the approach vector pipeline using the simple grasp generator on a bottle finally constrained to a side grasp. Each generated approach vector is represented by a CAD model of the hand.

filter service removes the AVs that are in collision with the environment, the object or the robot itself. The *orientation filter service* deletes the AVs that are not pointing to a determined direction ± a threshold. The *distance filter service* filters out the AVs that are further than a threshold from a configurable frame. An example of the AVs filtering process is depicted in Fig. 6.7.

6.3.6 Manipulation primitives

As detailed in Chapter 3, manipulation primitives are single reactive controllers designed to perform a specific primitive action on a particular embodiment. The primitives are configurable to adapt their behaviour to a specific task or environment. The primitives available on the system are briefly described below, a more detailed description with all the available parameters can be found in Chapter 3.

- Move arm to: can either use the planning or inverse kinematics services to move the selected arm to the desired position. Monitors the motion and stops if a collision is detected.
- Robust grasp: performs a reactive sensor-based grasp strategy with the selected arm and fingers.
- Hold: controls the fingers of the hand to keep the desired force.
- Lift: moves the arm upwards (w.r.t. world) until the desired lift distance is reached.
- Place: moves the arm towards the target supporting surface until contact is detected.
- Release: opens the hand to the desired opening position. Meanwhile zero-force control is used for a compliant release movement.
- Slide: slides the object over the surface towards the target position controlling the applied force.
- Push and pull: pushes or pulls an object compliantly for the desired distance or until the force limit is detected.
- Unscrew: grabs a threaded cap and performs the required twist movements pulling the cap until it is free for removal. While the cap is being unscrewed, the object needs to be fixed (e.g. by the other hand).
- Touch: moves the arm forward until a contact is detected.

Beyond manipulation primitives, there are other primitives needed to compose a task. Those auxiliary primitives can activate or deactivate perception pipelines, change parameters, introduce delays or wait for key strokes.

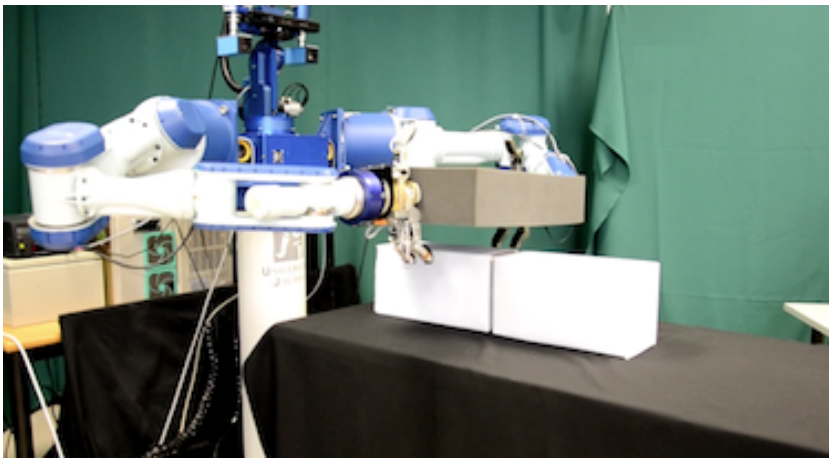


Figure 6.8: Dual arm manipulation of big objects.

6.4 Other research experiments

In addition to the experiments detailed in previous chapters, regarding object manipulation, contact perception, contact prediction and object detection and recognition, the architecture presented in this chapter was also used to perform other manipulation experiments. Dual arm control is another of the ongoing research lines that have been conducted with our robot. As depicted in Fig.6.8, some preliminary results have shown dual arm coordination abilities for manipulating big objects with both arms.

Besides grasping and manipulation experiments, the head of the robot has also been used for research experiments about biologically inspired learning. Work about implicit mapping of the peripersonal space can be found in [Antonelli et al., 2011] and more results of saccadic adaptation and learning saccade control are available in [Chinellato et al., 2012, Antonelli et al., 2013, Antonelli et al., 2015].

6.5 Implementation on other embodiments

Theoretically, the implementation of the layered architecture allows the software components to be used on different hardware platforms. The only modules that need to be implemented are the drivers and hardware interfaces. Thus, services, primitives and tasks can be used independently of the underlying hardware. However, this is not frequently the case, and services, primitives and even tasks are usually platform dependent and require modifications in order to adapt to other platforms. Although it is possible to implement services and primitives in a hardware agnostic manner, each platform has its specific features and not exploiting them would cripple the capabilities of the robot.

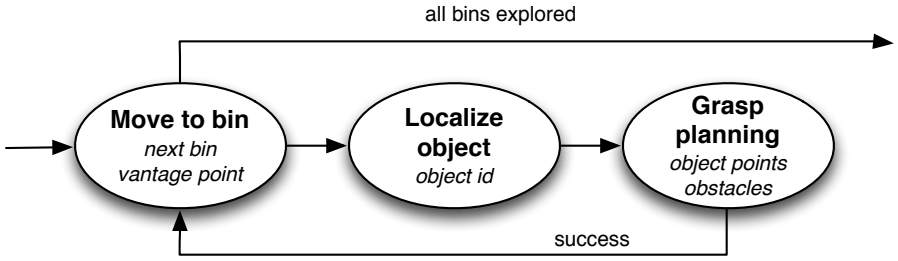


Figure 6.9: Exploration task definition for the APC. Parameters required by each primitive are written in italics. The target bin is switched by the task each time the Grasp Planning primitive finishes and before the Move to bin primitive is activated again.

All the modules presented in this chapter were implemented for Tomabtossals. Moreover, the four building blocks presented through this thesis to build a complete contact-based manipulation system were mainly implemented for the same platform. In order to participate in the APC (Amazon Picking Challenge), some of the modules developed for Tombatossals had to be ported to Baxter (see a detailed description of the robot in Appendix A.4), which was the robotic platform to be used during the competition. The aim of the challenge is to pick all the objects in an order list from a shelf and place them into a bin next to the robot. The robot is given a file with the target objects and it has to autonomously search, pick and place them into the bin.

To solve the APC task, we divided the problem into two sub-tasks: exploration and manipulation. The former uses the Kinect attached to the robot forearm to explore the shelf, look for the target objects and generate an approach vector to grasp each target object (see Fig. 6.11). The latter uses the list of plans generated by the exploration subtask to sequentially grasp the target objects and place them in the bin. Both subtasks are described using the manipulation primitives paradigm and depicted in Fig 6.9 and Fig 6.10.

The modularity and structure introduced by the software architecture made the adaptation of the services and primitives easier. On the other hand, the task, some primitives and services were implemented ad-hoc for the challenge. The visual and grasp planning process is depicted in Fig. 6.11, where the segmentation, clustering, recognition and grasp planning steps are depicted. Finally, Fig. 6.12 shows a picture of the robot grasping an object from the APC shelf and the result of the visual and grasp planning pipelines.

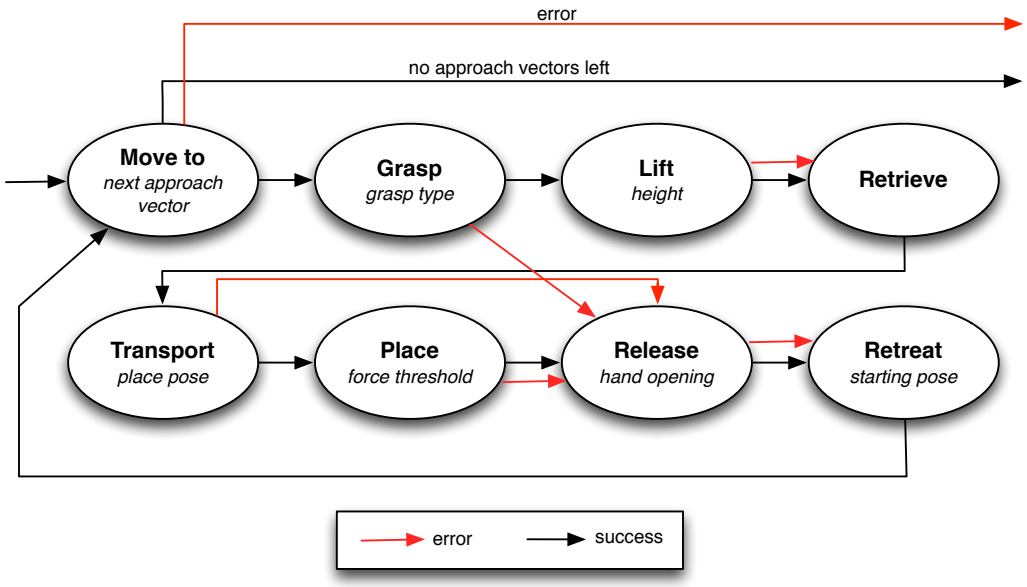


Figure 6.10: Manipulation task definition for the APC. Parameters required by each primitive are written in italics. Each primitive is configured before activating it. The *approach vector* and *grasp type* are obtained from the list of plans, the other parameters are defined in a configuration file and do not depend on the plan or the bin.

Figure 6.11: Grasp planning process for the APC challenge. Left: Initial image with bin reference frame. Middle: object segmentation, clustering and recognition. Right: grasp planning.

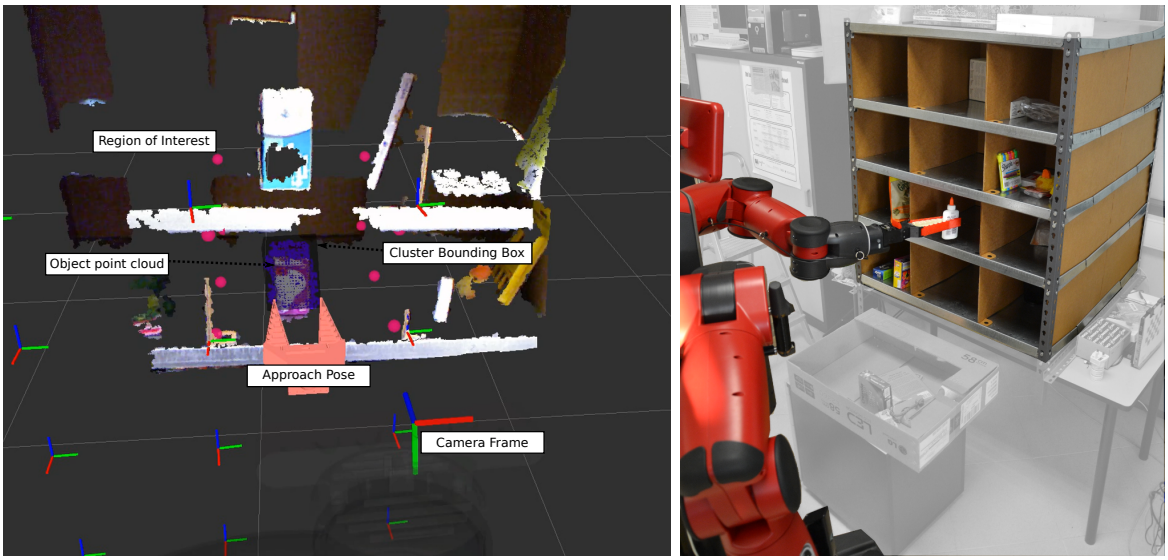


Figure 6.12: Left: Result of the visual and planning pipelines after the exploration of a bin of the APC shelf, the approach vector is represented by a CAD model of the gripper. Right: Execution of a manipulation task on a target object inside a bin of the APC shelf.

6.6 Conclusion

In this chapter, we have presented the layered software architecture used to integrate the proposed building blocks of the complete contact-based manipulation system.

Aside of the human inspired components identified in Chapter 2, there are other underlying abilities that are important and necessary for autonomous robotic manipulation, such as path planning, motor control, object perception or kinematic chain solvers. In this chapter we have also shown the basic skills that enable us to build more complex abilities on top of them. The work presented in this chapter, does not only include the architecture concepts and the implementation of some basic abilities. It also provides a tool to automatically generate code stubs for new modules, easing the creation of modules, increasing the readability of the generated code and saving a lot of programming and configuration time.

However, there are several drawbacks that need to be addressed in future versions of the architecture. Regarding message synchronization, when using a pipeline, it is not possible to determine which input generated which output. This is very important for example for learning algorithms. The design of pipelines, uses streaming data from a source and processes each message. Synchronization tools that allow the user to tag messages or to know exactly which message originated which output are an idea to

solve this issue. Other solutions such as a list of timestamps or synchronous pipelines are possible ideas that could solve this issue as well.

The process to set up a new experiment is complex and time consuming. First, all the involved modules have to be identified. Second, the configuration files for each of the modules has to be completed and the module to module connections have to be detailed. A graphical tool would be very useful in order to automatically generate the configuration files that properly connect modules to each other. Such a tool could rely on task specifications similar to the ones used in this thesis to generate the required task description and configuration files. That would complete the layered architecture allowing us to program at any abstraction layer, from low level drivers to high level graphical programming like Aldebaran's choregraphe[3] or Scratch[4].

Shared resources management is another important issue that has to be addressed. It is possible that different primitives use the same services or that two services use the same hardware interface. In the current version of the architecture, the developer has to be aware of this issue when setting up a new task. However, tasks will eventually become more and more complex and it will be very difficult to handle the shared resources manually. Thus, a mechanism to handle this issue has to be provided. A possible solution can be a warning system that automatically checks the modules used by the top level task.

The software architecture presented in this chapter was published together with the Tombatossals description in [Felip et al., 2015]. Moreover, the architecture and some of the abilities presented, were used by the RobInLab team during their participation in the 2015 Amazon Picking Challenge. Without the structure and organization of the architecture it would have been impossible to migrate all the important skills from Tombatossals to Baxter in only 4 months and be able to obtain good results at the lab and participate in the competition.

[3]Choregraphe: `www.aldebaran.com/en/robotics-solutions/robot-software/development`
[4]Scratch programming language: `scratch.mit.edu`