2013 10th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)
October 31-November 2, 2013 / Ramada Plaza Jeju Hotel, Jeju, Korea

An Open Teleconference Toolkit for Robotics

Stefano Ditrani[1], Fabrizio Caccavale[1], Anara Sandygulova[2] and Mauro Dragone[2]
[1] Department of Engineering, University of Basilicata, Potenza, Italy
(E-mails: ditranistefano@gmail.com, fabrizio.caccavale@unibas.it)
[2] School of Computer Science and Informatics, University College Dublin (UCD), Ireland
(E-mails: anara.sandygulova@ucdconnect.ie, mauro.dragone@ucd.ie)

Abstract *- This paper illustrates a new, open source toolkit enabling the seamless integration between robots and popular Internet-based teleconference systems. The toolkit has been designed to leverage a number of standards and to be as open and extensible as possible. This paper describes the rationale for the design of the new toolkit, and illustrates its implementation and its application to two popular robot platforms.*

Keywords - Robotic toolkit, remotely operated robots, robotic telepresence.

1. Introduction

Many robot applications combining cheap robotic devices with popular Internet-based teleconference systems, such as Skype and Google Talk, have emerged in recent years. Use cases vary from enabling users to call and remotely operate their toy robot to check their own homes while they are away [1], to more sophisticated tele-care robot systems [2] that can be used remotely by people to set-up video calls to their loved ones (as exemplified in Fig 1).
At the same time, progress in standardization of robotic software systems, such as the one pursued by the popular robotic operating system (ROS) initiative [3], opens up new opportunities for their successful integration with mainstream teleconference systems. Current software solutions in robotics are often of a component-based software engineering genre and provide a number of mechanisms and methodologies that can be used for the design, development and the execution of modular system architectures in terms of loosely coupled and potentially distributed components.
Contrary to past efforts, which have been tied to particular teleconference systems, specific robot frameworks and/or specific applications, we have designed an highly modular, and thus open, extensible and portable toolkit. We focused on supporting a number of different use cases, considering, for instance, both calls initiated by the robot and calls initiated by human users. In addition, we chose a number of standards and mainstream software engineering techniques in order to produce an easy to use and multi-platform toolkit.
The remainder of the paper is organized in the following manner: Section 2 provides an overview of both emerging teleconference and robotic standards, and discusses some of the related work. Section 3 presents the design and the implementation of our toolkit. Section 4 illustrates its use applied to two representative robotic platforms, namely, a Turtlebot robot driven by ROS software, and a Nao robot. For both platforms, we have used our toolkit to implement a number of illustrative applications compatible with Google Talk.
Finally, Section 5 summarizes the contributions of this paper and points to some directions to be explored in futureresearch.

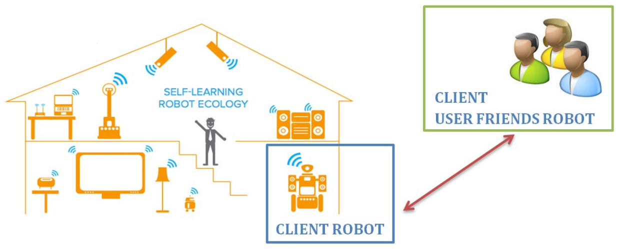

Fig. 1. Robots integrated with an Internet-based teleconference system

2. Background and Related Work

2.1 Teleconference technologies

Many instant messaging (IM) and Voice over IP (Voice over Internet Protocol) technologies exist today, among the most famous: Skype and Google Talk.
Skype is certainly the most familiar application for teleconferencing. Skype was released in 2003 as a Windows application but today it supports Mac OS and Linux as well as a wide range of mobile devices. Skype can be integrated with other applications through a public APIs that has varied considerably over time. Its Skypekit allows Internet-connected devices or applications to offer Skype voice and video calls. However, access to the Skype developer program (a per-requisite to use the kit) is limited and the runtime is only available for desktop platforms.
Google Talk was first released by Google in 2005. Users are required to activate a Google Account, after which they can start teleconferences from their computers, from Google+, or from their web-mail account. Unlike other instant messaging systems, Google Talk uses an open protocol: The Extensible Messaging and Presence Protocol (XMPP). In May 2013 Google launched the new messaging service Google Hangouts.
XMPP is an open-standard communications protocol for message-oriented middleware based on XML. The protocol has been developed for near real-time, instant messaging(IM), presence information, and contact list maintenance, but it is designed to be extensible. To this end, XMPP leverages TCP or other transport protocols (e.g. HTTP) to manage XML streams among remote clients. Each client is uniquely addressable by an address called JID. One of the key strengths of XMPP is that, unlike multi-protocol clients, it provides client

connectivity via special gateway services running alongside an XMPP server. The result is an highly decentralized architecture (similar to that of the Simple Mail Transfer Protocol, SMTP): Everyone can create their own XMPP server and integrate it with the rest of the network, thus giving the opportunity for individuals and organizations to have control over their communications. Since it was first introduced, back in 1997, tens of thousands of servers have been activated on the Internet today, and millions of people use XMPP through public services such as Google Talk. XMPP supports authentication and encryption standards. Finally, its use of XML and the availability of a number of client implementations makes it easily extensible. It has been used to support, among others, group chats, network management applications, collaboration tools, file sharing, gaming, and also remote systems control and monitoring.

2.2 Related work

There are many example of hobbyist robotic kits featuring teleconference/telepresence functionalities. Johny Lee's low-cost robot [4], for instance, uses a netbook mounted on top of an iRobot Create platform [5] and the Skype's Skype4COM windows-only desktop API [6]. A dedicated software component on the robot side listens for drive commands sent over the Internet, by using an additional communication channel to the one used by Skype, whose teleconference service is exploited. This type of solutions results in increased complexity, lower robustness and potential security problems. Sparky, and the newer Sparky Jr. projects [7] are open source projects based on a Skype plug-in, which is used to interact with Skype and to parse incoming text from the chat with a remote user. Some of the text, which is recognized and interpreted by the plug-in as control instructions, is routed to the motor controller software linked to the robot hardware. The remote user needs nothing more than the standard Skype client to call and control the robot. However, the resulting system is largely dependent on the specific robot API, thus effectively reducing its portability.
More sophisticated systems have been produced by industrial and/or research projects. For instance, the iRobot AVA is a mobile robot base with an extensible 'neck' for a head, which is also equipped with a small LCD screen and two cameras , one for telepresence and human interaction and another to assist an operator remotely driving the robot. The system does not use standard IM or Voip clients, but the remote user can operate the robot (to move it forward, backward, left and right), by availing of a graphical user interface (GUI). Similarly, Giraff [8] is a wheeled mobile robot designed to facilitate elderly people in their contact with their relatives, friends, and carers. Giraff is the focal point of two major EU grants, namely: (I) ExCITE [9], an AAL project that studies the Giraffe employed in ambient assisted living (AAL) application in three countries, (ii) and "Giraff+" [10], a project that explores how the Giraff can be part of a larger home system that provides increased levels of care for elderly people as their care needs grow over time. Giraffe s is based on the operating system Windows XP Embedded, and it is controlled remotely via the Giraff Pilot application, which allows remote operators to generate video calls and to pilot the robot.
The NAO Messenger application [11] is a Google Talk client for the NAO humanoid robot for Aldebaran [12]. The application uses the Nao to let users know when their friends are connected in Google Talk. The system, in its BETA version, is highly dependent from the Nao robot and does not offer live chat functionalities, as it requires the user to record a message before sending it to the Nao.
The system that shares more similarities with the one presented in this paper is [13]. Specifically, the system implements an architecture for robotic telepresence and teleoperation based on ROS and Skype. This allows a remote user to not only interact with people near the robot, but to view maps, sensory data, robot pose and to issue commands to the robot's by using a joystick. ROS provides a robot with the ability to localize itself and navigate with respect to a map, so the goal of the project is to share the robot's state with the remote user, and to accept commands from the user that are referenced to the map, all over Skype. The development is focused on text chat control using a standard Skype client; and map-based control using the Skype development environment (Skypekit).
Integrating Skype with external software is possible but this raises a technological issue since Skype's best known integration tools are for Windows whereas most robot platforms using ROS run Linux.

3. Toolkit Design

The system described in this paper is designed to address (i) system portability, in order to support multiple robot frameworks and easily fit different teleconference protocols, and (ii) extensibility in terms of use cases, for instance, to support both chat-based and speech-based user-robot interaction. Our goal is to give greater autonomy to the robot in order to allow a wide range of applications. Contrary to the systems described in the previous section, which tend to give greater control and capacity to the user, our system is designed to allow the robot to leverage its autonomous decision process, to initiate calls and/or react to user input, for instance, via speech-based interfaces. The system is designed to be extensible to new voice inputs and to any new services offered by the robots it operates with.

To this end, we distinguish between two sub-systems (see Figure 2), respectively:

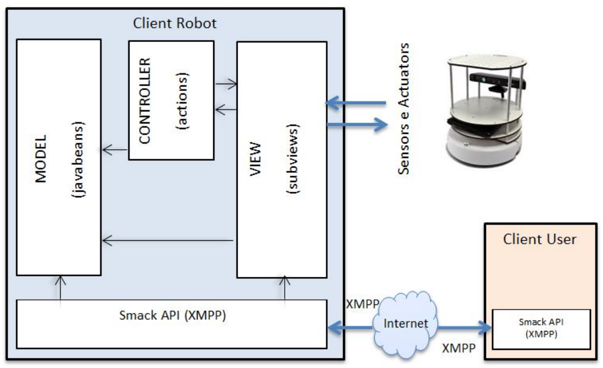

Fig. 2. System architecture

1. A ***robot-end*** sub-system, installed on the robot, which handles the interaction between the robot and the teleconference protocol. Given its advantages over similar mechanisms, as outlined in the previous section, our current system is based on XMPP.

2. A ***user-end*** sub-system, which handles the interaction between an external user graphical interface (GUI) and the XMPP protocol. At both ends, connection with XMPP is supported thanks to the Smack API [14], a pure Java library that can be embedded into a Java-based application to create anything from a full XMPP client to simple XMPP-based message notification.

At the robot-end, Google Guice [15] is used as a framework for dependency injection. This is a programming style in which dependencies between objects and/or system's components are not rigidly defined at compilation time (e.g. via explicit references to object and/or component implementations), but at the time the application is initialized, when they are "injected" into the collaborating parts thanks to specific framework mechanisms. Consequently, the resulting software systems are better equipped for the support of the application composition phase where components are initialized and bound in different ways or re-used for different application. Google Guice is an open source software framework for the Java platform released by Google under the Apache License. It provides support for dependency injection using annotations to configure Java objects. For the purpose of our system, the use of a dependency injection mechanism such as Google Guice offers the following advantages:

- it improves its re-usability in conjunction with different robot frameworks;
- it eases unit and integration testing, as it is possible to inject mock implementations of component's dependencies

The robot-end sub-system is a Java application composed of components responsible for the handling of the XMPP protocol for instant messaging, and components interacting with the specific robot platform employed in the application.

The sub-system has been designed according to the *Model-View-Controller* (MVC) architectural pattern [16], which aims to decouple the responsibility of the individual components, and to separate the part relating to the application logic, which are handled by the Controller, from the application status, handled by the Model, and its presentation, handled by the View. The latter is usually used to manage user-system interaction through a graphical user interface (GUI).

We applied the MVC design pattern to our robot-end sub-system, which does not have a GUI, but that assumes similar input/output responsibilities for the robot system and the teleconference protocol. Specifically:

- The ***Model*** is based on JavaBeans Java technology and it is composed by a number of classes that collectively represent the application domain and the application logic. This includes, for instance, classes representing a history of past teleconferences, and contacts of "friends" of the GoogleTalk account used by the robot. The *Model* class in our implementations offers a centralized storage of all the models defined for a specific application, and provides a number of utility methods to help managing their life cycle.

- The ***View*** consists of a set of classes, called *sub-views*, that are responsible for interacting with the external environment, processing requests from the user located with the robot (e.g. via speech-based or other interfaces); from components internal to the robot (e.g. belonging to the robot's control system), but also from remote robots or remote users (through the XMPP protocol). For instance, a sub-view is used to manage local speech-based input, in order to receive and process instructions uttered by the human near the robot, while another sub-view is used to manage speech synthesis output, in order to give speech-based feedback. Other sub-views are used to trigger specific robot services, from basic movements, such as *move left*, *move right*, to more sophisticated behaviours that may be implemented by the specific robot framework, such as *find user*, *clean room*, etc. While the majority of the sub-views must be specialized to provide robot-specific sub-views, we have defined a specific sub-view, called *IMClient* to interact with the specific IM protocol used by the application. Such a sub-view is a robot-agnostic sub-view that manages the interaction between the other sub-views and the specific IM protocol, by listening to incoming IM packets, and by letting other local sub-views to transmit IM packets to remote users.

- Inputs generated by the sub-views are notified to the ***Controller***, which implements the control logic. The Controller is based on the Command design pattern [17]: It includes a number of *Action* classes whose methods are executed when the robot receives specific instructions, and a *Controller* class mapping each instruction to its corresponding action. Once an input is received by the Controller, this consults and updates the Model before deciding the proper action that needs to be executed by the application. Each Action component holds a reference to a sub-view, whose method it invokes in order to trigger the achievement of a specific robot's operation.

4. Examples and Tests

As discussed in the previous section, the MVC patterns allows us to fit different robot technologies by re-using our implementation of the Controller and the Model, and by

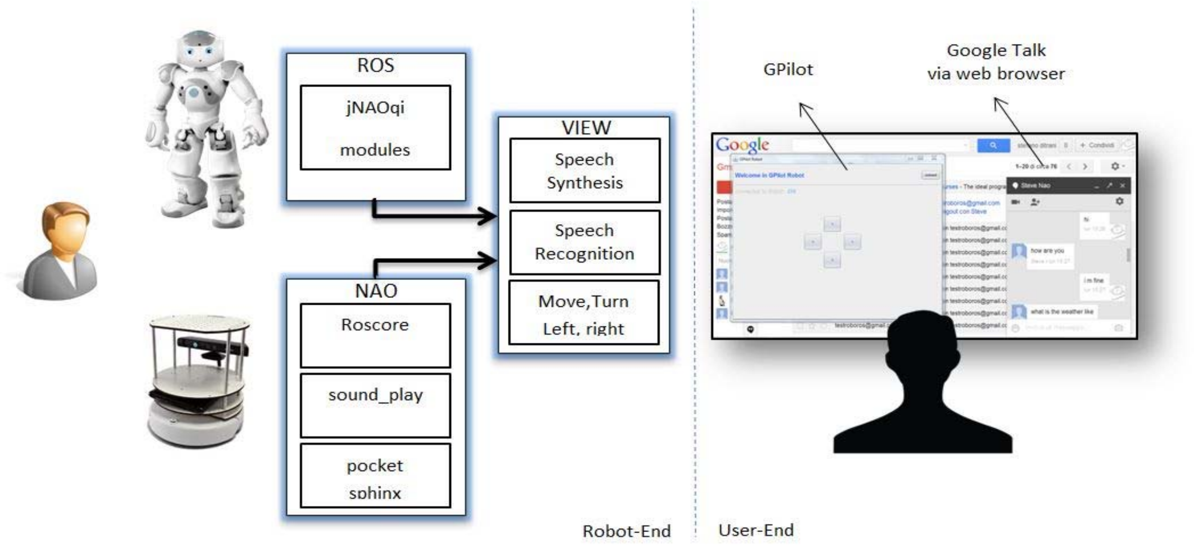

Fig. 3. System for multiple robot platforms and application User-End.

addressing any robot-specific and protocol-specific issue in the View.

In order to demonstrate how our system can easily support multiple robot platforms and use cases, we have created two applications by implementing two distinct implementations of the sub-view classes defined in our architecture. Specifically, our examples support: robots operated via ROS software and Nao humanoid robots.

Both implementations are based on Google Talk, which supports the XMPP protocol. However, it is sufficient to modify a parameter in a configuration file provided with the application, to alternatively use another IM client with XMPP support. In order to validate the resulting systems, we have registered a Google Account for each robot.

At the user-end, both application interact with remote robots (and with the remote users co-located with those robots) via a standard Google Talk web client, which can be used to see which robots are online, and to initiate teleconference calls and/or chat sessions. In addition, we have developed a Gpilot application that is easily integrated with the same web browser in which the Google Talk client operates. The Gpilot is a Java-based GUI that uses the Smack API to communicate via XMPP with the robot-side of our application, once a teleconference is initiated through the Google Talk client. The Smack API allows us to use the serialization features of the XMPP protocol, as it provides an easy mechanism for attaching arbitrary properties to XMPP communication packets. Each property has a String name, and a value that is a Java primitive or any Serializable object. In this manner, the Gpilot can exchange data and instructions with the robot-side of the system, without having to piggyback on the standard chat stream, as in some of the systems we have reviewed in Section 2.2. (see Figure 3)

4.1 ROS Application

In order to support ROS-based systems at the robot-end, we avail of ROSJava, the first pure Java implementation of ROS.

In ROS, generally the nodes are synonymous with processes. In ROSJava, however, nodes runs within a single process, i.e, the Java VM, from where they can communicate with any other ROS node (e.g. with the nodes installed on the robot) through a publish-subscribe communication pattern mediated by the roscore server.

Our ROS implementation of the robot-end's view is composed by a number of ROSJava's *NodeMain* classes, each encapsulating a ROS node used to specialize a single sub-view in the architecture outlined in the previous section.

We have the following ROS-enabled sub-views:

- *Sub-view SpeakerROS*: Publishes the text to be uttered by the robot on the topic *"speaker"*. For the actual speech synthesis, we rely on the speech and *audio_common* packages, two third-party ROS packages that must be installed on any of the robots operating with our system. However. in order to decouple our sub-view from the actual robot system, we also provide a *Sound* node, which subscribes to the "*speaker"* topic, and executes the appropriate instructions from the underlying speech synthesis implementation. Different implementations can be fitted by simply providing a different version of this node.
- *Sub-view ListenerROS*: Subscribes to the topic: *" /recognizer/output"*, which must be used by a speech recognition software to report speech uttered by the local user to the robot. Specifically, our current implementation relies on the ROS interface to the pocketsphinx speech recognition system [18], which must be pre-installed on the robot. Currently, this feature is used to enable the local user to interact with Google Talk by uttering basic instructions, in order to query the list of online friends and start/stop a teleconference (i.e. "is <contact> online?", "Call <contact>", "Bye"). To this end, we have provided a short sphinx grammar that models the set of instructions to be recognized by the system. Finally, the same feature is also used to transmit the free speech of the local user (as opposed to specially recognized instructions) via the chat system.
- *Sub-view MoveROS*: Publishes the topic *"/cmd_vel"*, which is a standard topic used to control the velocity of the robot. Such a functionality is provided as an example to demonstrate how remote users can be allowed to control robots via the XMPP protocol. While this example enables

remote users to direct control of the robot platforms using our systems, more sophisticated control schemes can be designed by simply extending the View sub-systems, as discussed in Section 4.3.

4.2 NAO Application

In oder to test our system with the Nao humanoid robot, we wrote a version of the sub-view classes by using *JNAOqi*, the Java interface to the *NAOqi SDK* [19]. Collectively, the resulting sub-views provide an interface toward the Nao's behaviour and input/output systems. Specifically, they allow our system to leverage the built-in Nao's speech interface capabilities, and also to activate, configure and deactivate existing Nao's behaviours via XMPP, by using the proxy classes that are included in the NAOqi SDK to give direct access to the capabilities of the Nao.We have the following Nao sub-views:

• the s*ubview SpeakerNAO* implements methods for the management of the Nao's speech synthesis features, by using the *ALTextToSpeechProxy* class.

• the s*ubview ListenerNAO* implements methods for the management of Nao's speech recognition features, by using the *ALSpeechRecognitionProxy* class. As for its ROS-based equivalent, we have provided a grammar to

• the s*ubview MoveNAO* implements methods to interact with the Nao's behaviour system by using the *ALBehaviorManagerProxy* class.

4.3 Putting it all together

For both robot frameworks, we have built a number of demonstrative applications by using a Nao robot and a ROS-based Turtlebot.

In these applications, a user can ask the robot to know which of her friends are connected in Google Talk, and start chatting with them through the robot. The user can also decide to initiate a mixed chat, by relying on the robot's speech-recognition capabilities to transmit only the corresponding text to the remote user, while the remote user will have her own text uttered by the robot.

Alternatively, a teleconference or a chat session may also be initiated by the robot, autonomously, on the basis of a pre-programmed routine, or in response to some event perceived thanks to its sensors. The robot's control system can act on the robot-side of our application to initiate a session with a remote user. Supporting such a use case can be useful, for instance, to automatically contact the user if this is outside her own home and something anomalous, such as an intruder or another emergency, is detected by the robot, or to inform a relative that an elderly user requires some form of assistance.

In addition, thanks to the Gpilot GUI, the remote user can ask the robot to perform simple movements or to activate simple services, for instance, to move to certain rooms, or to follow its user, if this is performing some activity while engaging in a Google Talk conversation.

While our current implementation provides only limited control of the robot platforms we have used in our tests, more sophisticated robot services can be easily fitted in our system. It is enough to extend the Gpilot application and the MVC pattern used at the robot-side to handle new control instructions and link them with any of the new functionalities that may be supported by the robot. Notably, such links are defined thanks to dependency injection mechanisms that allow us to drastically reduce the interventions to our code-base that we need to perform in order to support new functionalities. Those interventions are mostly restricted to changes to the configuration file used to configure the two ends of our application.

4.4 Usage Example

In order to more concretely illustrate the ease of use afforded by the new toolkit, this section gives more details on one of the example applications we have implemented. Specifically, we focus on a security service in which we have programmed our Nao robot to contact its users whenever an intruder is detected while they are away from home. Figure 4 shows part of the code for two new components that must be implemented to support such a use case, respectively: (i) a new *AlarmNao* sub-view with a *IAlarmNao* interface, and (ii) its corresponding action class *ActionAlarm,* implementing the generic *IAction* interface.

```
@Singleton
public class AlarmNAO implements IAlarmNAO {
  @Inject
  private Controller control;

  //implement methods defined in the interface
  public void alarm () {
     //use Nao framework to detect intruder
     . . .
     if(intruderIsDetectd) {
     control.getAction(“ActionAlarm”).execute();
     }
   }
}
```

Fig. 4. Part of the new subview class supporting the alarm Nao use case.

The implementation of the *alarm* method in the *ActionAlarm* sub-view is responsible for interacting with the Nao behaviour framework and to recognize possible situations that may signal the presence of an intruder, for instance, by using the Nao's sonar sensors and the Nao's face recognition capabilities (this part of the code, which is specific to the Nao robot, is not included in the pseudo-code showed in Figure 4).

If a possible intruder situation is detected by the Nao, the *AlarmNao* class invokes the execution of the *ActionAlarm* action. Noticeably, the reference to the controller (the control variable used in the code) is injected by using the Google Guice framework and the developer of the *ActionAlarm* class is completely shielded by the implementation details of the alarm action.

```
@Singleton
public class ActionAlarm implements IAction {
  @Inject
  private Model model;
  //implement methods defined in the interface
  public void execute () {
     String text = “Allarm, Help me!”;
     User user = (User) model.getBean(“User”);
     user.sendMessage(text);
   }
}
```

Fig. 5. Part of the ActionAlarm class

Figure 5 shows part of the code of the ActionAlarm class. Specifically, it shows how the implementation of its *execute* method uses our toolkit to send a message to the user. The *sendMessage("text")* routine in the *User* class allows developers to send a chat message or email through the XMPP protocol.

Noticeably, in order to implement an equivalent service for robots based on the ROS framework, such as our Turtlebot, developers only need to implement an alternative ROS-based version of the alarm sub-view (e.g. *AlarmROS*). Such an implementation will use Turtlebot's sensors, i.e. its Kinect 3D camera, to monitor for intruders, but re-use all the other classes already implemented to support the security service.

5. Conclusion and Future Work

In this paper we have presented an open source toolkit enabling the seamless integration between robots and popular Internet-based teleconference systems. Our toolkit is designed to be as flexible as possible, in order to easily support multiple use cases, multiple robot platforms and multiple IM protocols.

Concerning the speech-based interaction, the performance of our systems relies on the state of the art available on the platforms we have tested, as well as on the particular robot's hardware. In particular, while we have run very successful examples of chat and remote robot control for both platforms, the Nao's capabilities for speech recognition are far superior to those we obtained with the Turtlebot robot. Future work will produce a ROS-based version that will use the full capabilities of the microphone array included in the Turtlebot's Kinect sensor.

We also plan to leverage the fact that both ends of our system are based on Java, to allow seamless integration of ROSJava components on the user-side. With our architecture, it should be relatively straightforward to create a backchannel system, which will use the IM protocol to initiate calls before letting both sides operate by exchanging ROS messages directly. In this manner, new extension to both service and GUI functionalities may be defined solely on the user-side and will not require any intervention to the robot-side.

We aim to evaluate the performance of such an approach in our future work, and also address security and performance issues likely arising from such a level of openness.

References

[1] F. Michaud, P. Boissy, H. Corriveau, A. Grant, M. Lauria, D. Labonte, R. Cloutier, M. Roux, M. Royer, and D. Iannuzzi, *"Telepresence robot for home care assistance"*, in AAAI Spring Symposium on Multidisciplinary Collaboration for Socially Assistive Robotics, 2007.

[2] H. Nakanishi, Y. Murakami, D. Nogami, and H. Ishiguro, *"Minimum movement matters: impact of robot-mounted cameras on social telepresence"*, in Proceedings of the 2008 ACM conference on Computer supported cooperative work, ser. CSCW '08. New York, NY, USA: ACM, 2008, pp. 303–312. [Online]. Available: http://doi.acm.org/10.1145/1460563.1460614

[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, *"ROS: an open-source Robot Operating System,"* in ICRA workshop on Open-Source Software, 2009.

[4] J. C. Lee. (2011) Low cost video chat robot v2. [Online].Available:http://procrastineering.blogspot.com.au/2011/02/low-cost-video-chat-robot.html

[5] (2011) Low cost video chat robot. [Online]. Available: http://youtu.be/9LNS9CivO34

[6] Skype4COM. [Online]. Available: http://dev.skype.com/accessories/skype4com

[7] (2012) Sparky jr project. [Online]. Available: http://sparkyjr.ning.com

[8] Giraff. [Online]. Available: http://www.giraff.org

[9] Excite. [Online].Available:http://www.oru.se/ExCITE

[10] GiraffPlus [Online]. http://www.giraffplus.eu/

[11] NAO Messenger. [Online]. https://store.aldebaran-robotics.com/product/nao-messenger

[12] Aldebarian. [Online]. https://http://www.aldebaran-robotics.com

[13] Peter Corke, Kyran Findlater & Elizabeth Murphy *"Skype : a communications framework for robotics"*, 2012

[14] Smack API. [Online]. http://www.igniterealtime.org/projects/smack/index.jsp

[15] Google Guice. [Online]. https://code.google.com/p/google-guice

[16] Trygve Reenskaug and James O. Coplien. *"The DCI Architecture: A New Vision of Object-Oriented Programming"*, 2009 [Online] Available: http://www.artima.com/articles/dci_vision.html

[17] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. "Design Patterns: Elements of Reusable Object- Oriented Software", Addison Wesley, 1995, ISBN 88-7192-150-X

[18] Ros-Pocketsphinx-speech-recognition[Online]. Available:https://code.google.com/p/ros-pocketsphinx-speech-recognition-tutorial

[19] NAOqi SDK[Online]. Available: https://community.aldebaran-robotics.com/doc/1-14/index.html