STAIR: Hardware and Software Architecture

Morgan Quigley and **Eric Berger** and **Andrew Y. Ng**
Computer Science Department
Stanford University
{mquigley, eberger, ang}@cs.stanford.edu

Abstract

The STanford Artificial Intelligence Robot (STAIR) project is a long-term group effort aimed at producing a viable home and office assistant robot. As a small concrete step towards this goal, we showed a demonstration video at the 2007 AAAI Mobile Robot Exhibition of the STAIR 1 robot responding to a verbal command to fetch an item. Carrying out this task involved the integration of multiple components, including spoken dialog, navigation, computer visual object detection, and robotic grasping. This paper describes the hardware and software integration frameworks used to facilitate the development of these components and to bring them together for the demonstration.

Introduction

At the AAAI 2007 Mobile Robot Exhibition, we presented videos of the STAIR robot performing a "fetch a stapler" demonstration. In this paper, we describe the hardware and software integration systems behind this demonstration. We found that having a consistent software framework was critical to building a robotic system of the level of complexity of STAIR, which incorporates components ranging from spoken dialog to navigation to computer vision to robotic manipulation. In this paper, we describe some of our design decisions, as well as lessons learned in building such a system. We also describe the specific technical details of applying these ideas to having the robot fetch items in response to verbal requests.

Hardware Systems

The first two robots built by the STAIR project were named simply STAIR 1 and STAIR 2. Each robot has a manipulator arm on a mobile base, but the robots differ in many details.

STAIR 1

This robot, shown in Figure 1, was constructed using largely off-the-shelf components. The robot is built atop a Segway RMP-100. The robot arm is a Katana 6M-180, and has a parallel plate gripper at the end of a position-controlled arm that has 5 degrees of freedom. Sensors used in the demonstrations described in this paper include a stereo camera, a SICK laser scanner, and a pan-tilt-zoom (PTZ) camera. A second SICK laser scanner was mounted on top of the robot on a panning motor, so as to obtain 3d point clouds of objects in front of the robot.

Figure 1: The STAIR 1 robot

Most of the sensors were mounted on an aluminum frame bolted to the table of the Segway base.[1] We did not use the self-balancing capabilities of the Segway. Instead, we constructed an additional aluminum frame which added a wheel to the front and a wheel to the back of the robot, so that it became statically stable. This was done as a practical

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] The aluminum frame was built out of parts made by 80/20 Inc. The other sensors used were a Bumblebee stereo camera, a Sony EVI-D100 PTZ camera, and a SICK LMS-291 laser scanner. A second SICK LMS-200 laser scanner was mounted on top of the robot, on a AMTEC PowerCube module.

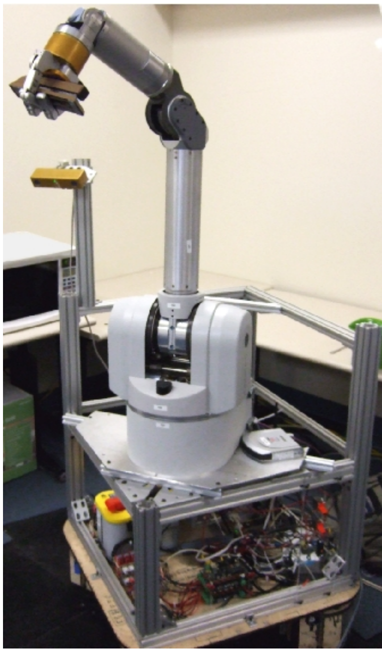

Figure 2: The STAIR 2 robot

measure to avoid damage in the event of an emergency stop, at the cost of increasing the footprint of the robot by a few inches.

The robot is powered by a deep-cycle 12-volt battery feeding an array of DC-DC converters, which produce the various DC voltages required by the robot's subsystems. An onboard automatic battery charger allows the 12-volt power rail to function as an uninterruptable power supply (UPS), which allows the robot's computers and sensors to remain running as AC power is removed for mobile experiments. The power system allows for approximately two hours of runtime at typical loads.

Onboard computation is provided by a Pentium-M machine running Linux and a Pentium-4 machine running Windows. These machines are connected via an onboard ethernet switch and, via an 802.11g wireless bridge, to workstations throughout the building.

STAIR 2

The STAIR 2 platform is shown in Figure 2. The wheeled base (comprising the bottom 10 inches of the robot) was designed and constructed by Reuben Brewer of the Stanford Biorobotics Laboratory. This base has four steerable turrets, each of which contain two independently-driven wheels. As a result, the platform can holonomically translate in any direction, turn in place, or translate and rotate simultaneously. Desired motions in the robot's coordinate system are translated by a dedicated XScale processor (on a Gumstix Verdex board) into motor commands.

This platform uses a Barrett WAM arm, which is mounted on an aluminum frame built on top of the wheeled base. The WAM arm is a force-controlled arm with seven degrees of freedom, and also integrates a 3-fingered hand. We used a dedicated onboard Linux PC to control the arm. The robot also has a Bumblebee2 stereo camera, and additional onboard computation is provided by a second Linux machine.

The power and networking systems are similar to STAIR 1: an ethernet switch connects the onboard computers, and a 802.11g wireless bridge provides connectivity with the (much greater) computational resources offboard the robot.

Software Systems

A significant part of our effort on STAIR involved designing and implementing a framework to support robot software development, so as to enable applications such as the "fetch a stapler" demonstrations. Many other researchers have worked in this area, producing notable robotics frameworks such as Player/Stage (Gerkey, Vaughan, & Howard 2003), CARMEN (Montemerlo, Roy, & Thrun 2003), MCA (Scholl, Albiez, & Gassmann 2001), Tekkotsu (Tira-Thompson 2004), Microsoft Robotics studio, and many others.[2] After investigating these existing frameworks, we determined that our platform and goals differed sufficiently from those of the designers of other frameworks that implementing a purpose-built framework would be worthwhile.

Requirements

Parallel Processing Our application runs a single, large, highly capable robot, and requires carrying out a considerable amount of computation. Our software has both hard- and soft-real-time constraints, and also carries out longer-running planning and scene analysis tasks. The onboard computational resources of the robot cannot support all the required computation, so we must spread the computational load across offboard machines as well.

Modularity Because the STAIR project involves dozens of researchers contributing to a sizable code base, it is important to enforce modularity between software components, so that components can be debugged and verified in isolation as much as possible.

Cross-Platform Most of our available computational resources are in the form of Linux workstations and server racks. However, a few of the robot's sensors came with only binary Windows drivers. Therefore, our system must run both Linux and Windows operating systems, and cross-platform communication is required.

Robot-independent Because we are running two robots which have completely different hardware, the software must be as robot-independent as possible. Some software modules function as device drivers and thus are tied to hardware. However, as many software modules as possible should operate only on hardware-independent abstractions, to limit the size and complexity of the code base.

[2]The web pages of these frameworks are:
Player/Stage: http://playerstage.sourceforge.net
CARMEN: http://carmen.sourceforge.net
MCA2: http://www.mca2.org
Tekkotsu: http://www.cs.cmu.edu/˜tekkotsu
Microsoft Robotics studio: http://msdn.microsoft.com/robotics/

Code Aesthetics As with any large software project, keeping code clean and streamlined makes research progress on the robot significantly easier.

Design Choices

To meet the aforementioned requirements, we built a library called *Switchyard*, which supports parallel processing through message passing along a user-defined, task-specific graph of connections between software modules.

Modularity is enforced through the operating system process model: each software module executes as a process on some CPU. The TCP protocol was chosen for message passing, because it is supported on all modern operating systems and networking hardware. Its operation is essentially lossless, which leads to simpler parsers that do not have to handle re-synchronization.

From an aesthetic standpoint, the library is in the form of C++ classes which each module extends to provide the required functionality. Networking, routing, and scheduling code do not show up in the software modules, as they are provided by superclasses. This allows most modules to have very little boilerplate code.

These design choices are certainly debatable. Indeed, we are currently designing another robotics software platform which builds upon lessons learned from the current framework. However, for completeness we will provide details of Switchyard's design and operation, as used in the demonstration videos shown at the AAAI 2007 Mobile Robot Exhibition.

Message-Passing Topology

Switchyard sets up a "virtual cluster" of computers on top of an existing cluster of networked machines. The term "virtual cluster" is meant to indicate that a subset of machines on the network will operate as a cohesive group during a run of the robot.

Master Server One computer in the virtual cluster is chosen to be the *master server*.

Importantly, the master server does **not** process all the traffic flowing through the virtual cluster. This would produce a star network topology, which could be highly inefficient for networks with heterogeneous connections between machines. As a concrete example, consider a STAIR robot with an onboard ethernet switch connecting several machines on the robot, and a wireless bridge connecting these machines to the building's network, which in turn consists of many ethernet switches connecting many more machines. Throughput between machines on either side of the wireless link is excellent, but throughput across the wireless link is often slow and variable-speed as the robot moves through the building. The master server must reside on one side of the wireless link, and if it were to process all data, the wireless link would grind throughput to a halt across the entire virtual cluster, particularly if there are data-heavy flows on the subnets on each side of the wireless link.

The master server, then, is only present to automate the startup and shutdown of the virtual cluster. Data payloads sent between software modules flow on peer-to-peer TCP connections. On startup, the master server loads an XML description of the desired connection graph (the topology of the "virtual cluster") and automates its creation, as described in the next paragraph.

Process Launcher A simple "process-launching" program runs on every machine that is a part of the virtual cluster. As command-line parameters, this program receives the IP address of the master server and its "virtual machine name" (which need not coincide with its IP host name). Then, the process launcher connects to the master server, announces its name, and receives back a list of processes that are to be launched.

This step is only for convenience; if a process needs to be launched manually (for example, inside a debugger), it can be excluded from the automatic-launch list.

Process Connection Switchyard processes are invoked with the IP address of the master server, a "virtual name" (which need not coincide with the executable filename), and an available TCP port number on which to open a server. Processes start a server on their assigned port, connect to the master server, announce their name, and receive back a list of other processes with which to establish peer-to-peer connections. The processes then automatically connect to their peers, and can start producing and consuming data.

Data flows Data flows in Switchyard are always unidirectional and asynchronous with respect to any other modules. The sender (or "upstream") node sends chunks of data whenever it is ready. Each data flow can have any number of receivers. Data always flows in chunks or "quanta" that are meaningful to downstream nodes and cannot be subdivided logically. Some examples of "quanta" are images, laser scans, maps, matrices, or waypoint lists.

Although each of these "quanta" could be divided into smaller units (i.e. images and matrices could be divided into rows or blocks), downstream nodes would likely have to reconstruct the original logical unit (e.g., image or matrix) before processing could begin. Thus, to reduce code size and complexity in the receiving node, *Switchyard* only sends an entire "quanta" at a time.

To save boilerplate code, the data flow model is written as an abstract C++ class. This abstract superclass contains all the networking and sequencing code required to transmit byte blocks of arbitrary size. The superclass is derived to create each type of data flow in the system. Data flows types currently in use include:

- 2D, 3D, and 6D points
- 2D waypoint paths
- Particle clouds (for localization)
- Images and depth images
- Grid-world maps for navigation
- Arm (configuration-space) coordinates and paths
- Text strings
- Audio snippets
- Miscellaneous simple tokens (for sequencing)

Each subclass contains only the code necessary to serialize its data to a byte stream and the code to deserialize itself when presented with its byte stream. These methods are implemented as C++ virtual functions, which allows the higher-level scheduling code to invoke the serialize and deserialize methods without needing to know what is actually being transmitted. This use of C++ polymorphism significantly reduced the code size and complexity.

Since the computation graph runs asynchronously, whenever a process is ready to send data to its downstream peers, it invokes a framework-provided function which does the following:

1. The serialize virtual method in the data flow subclass fills a buffer with its byte-stream representation.
2. The data flow subclass reports the length of the byte stream representation of its quanta.
3. The byte-stream size is sent downstream via TCP.
4. The byte-stream itself is send downstream via TCP.

On the downstream side of a data flow, the framework does the following:

1. The byte-stream size is received and adequate space is allocated, if necessary, to receive it.
2. The byte-stream itself is received and buffered.
3. The deserialize virtual method in the data flow subclass re-creates the data structures.
4. A virtual function is called to notify the receiving process that the data structures are updated.

To avoid race conditions, each data flow has a mutex which is automatically locked during the inflation and processing of each data flow quanta. Thus, the data-processing code does **not** need to be re-entrant, which can help simplify its structure. The framework will silently drop incoming quanta if the process has not finished handling the previous quanta.

Data Flow Registration In the initialization of a Switchyard process, data flows must be instantiated and registered with the framework. This is typically done with a single line of C++ code (for each data flow) in the process constructor.

If the process will **produce** data on this flow, the following actions are taken by the framework:

- The data flow name is copied into framework data structures which will route incoming TCP connections which wish to subscribe to this flow.
- As other (receiving) processes connect to this (sending) process, the socket handles are stored with the data flow requested by the incoming connection.
- When this (sending) process wishes to send data on this flow, the framework calls the "deflate" virtual method (as described in the previous section), and sends the resulting byte stream to any and all active TCP connections for this flow.

If the process will **receive** data on this flow, the following actions are taken by the framework:

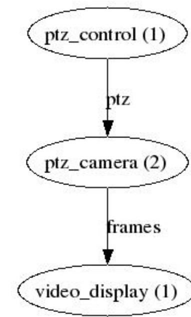

Figure 3: Pan-tilt-zoom (PTZ) camera control graph

- A thread is spun off to handle the data flow.
- This thread attempts to connect via TCP with the process that produces the data flow.
- Once connected, the thread announces the data flow it wishes to receive.
- The thread parses the data stream and invokes user code to process each incoming quanta, as discussed in the previous section.

By organizing the behavior in this manner, the entire peer-to-peer connection scheme can be completed automatically by the framework. This saves a great deal of repeated (and bug-prone) networking and sequencing code in each process.

Configuration Ports Many robotics software modules have startup parameters. For example, a map server has access to many map files, a laser scanner can be configured in a variety of resolutions, and so on. Following the precedent set in the Player/Stage framework, the graph XML file itself can optionally contain startup parameters which, at runtime, will override the default values hard-coded in each process.

Operation

To run an experiment or demonstration, the following steps must occur:

Graph Design The machines available to run the experiment must be described in the graph XML file, with either their hostname or IP address. Next, the software processes needed for the experiment or demonstration must be selected or written. After listing the machines and processes, the connections between the processes must be defined in XML.

As a concrete example, the following graph XML file routes video from one computer to another and allow remote pan-tilt-zoom (PTZ) control:

```
<graph>
  <comp name="1">
    <proc name="video_display">
    <proc name="ptz_control">
  </comp>
  <comp name="2">
    <proc name="ptz_camera">
      <port name="compression_quality" value="40"/>
    </proc>
  </comp>
  <conn from="1.ptz_control.ptz" to="2.ptz_camera.ptz">
  <conn from="2.ptz_camera.frames" to="1.video_display.frames">
</graph>
```

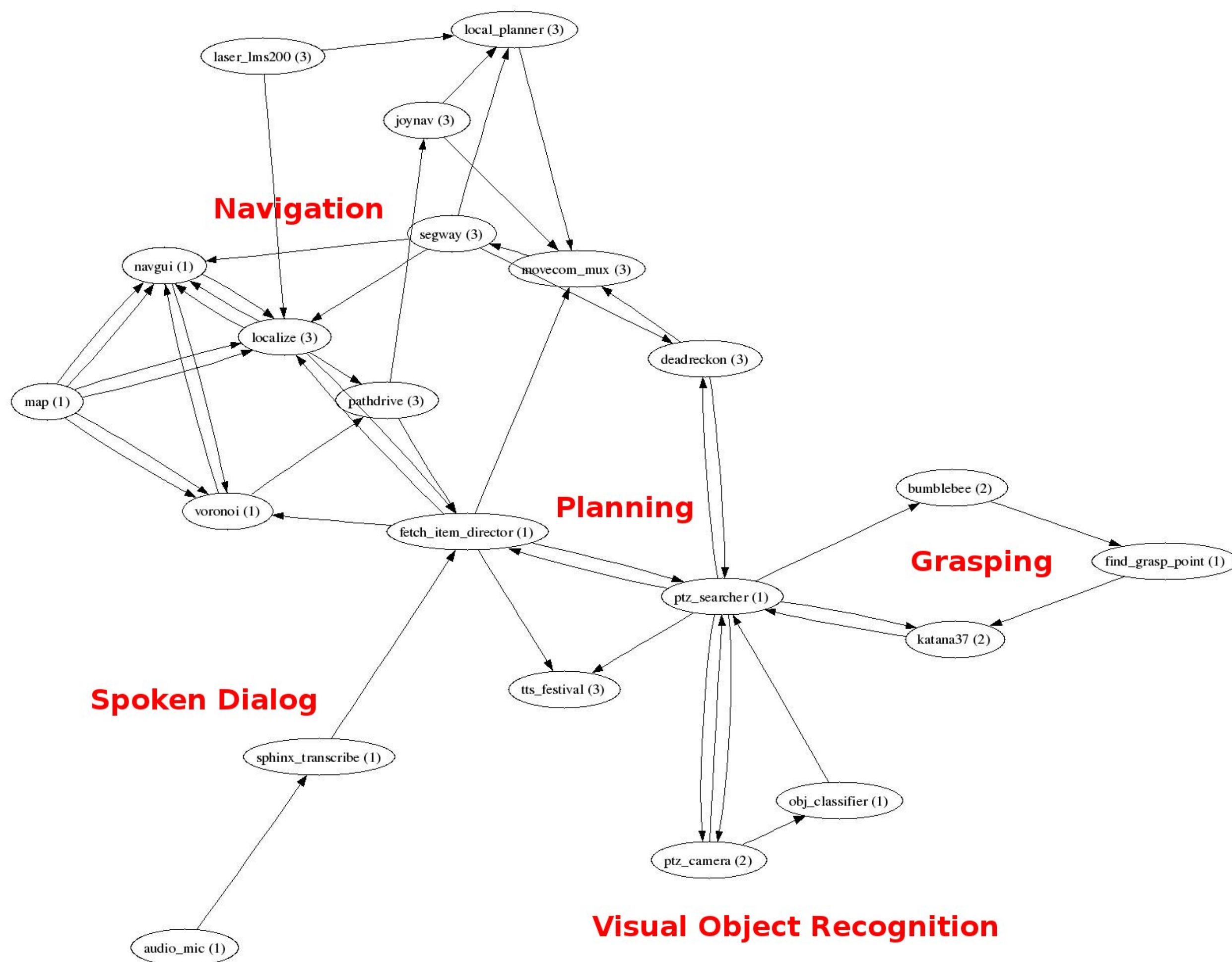

Figure 4: Graph of the "fetch a stapler" demonstration. The large red text indicates the tasks performed by various regions; it is not a functional part of the graph.

When this graph is run, it will cause two processes to start up on computer 1, and one process to start on computer 2. Once the processes are up, they will connect to each other automatically and data will start flowing. A visualization of this simple graph is shown in Figure 3. It allows computer 1 (typically off the robot) to view the video stream and command the camera to pan, tilt, and zoom. The camera device driver is running on computer 2 (typically on the robot) and the video is transmitted across the network as individual JPEG-compressed images.

The "fetch an item" demonstration involved a much larger graph. As shown in Figure 4, this graph involves 21 processes running on 3 machines, two onboard the robot and one offboard. The modular software structure allowed the (parallel) development of many of these modules in much smaller, even trivial, graphs. Unit testing in this fashion reduced development time and helped to isolate bugs.

The asynchronous nature of this framework is also shown in Figure 4. Cycles in this graph would create potential deadlocks if the graph were to operate synchronously. To keep things simple, the framework enforces no synchronization: any and all synchronous behavior is implemented using local state variables.

Framework Usage: "Fetch a Stapler"

This section will discuss how the software and hardware systems were used to perform the "fetch an item" demonstration. In this demonstration, a user verbally asks the robot to fetch an item (a stapler). In response to this spoken command, the robot navigates to the area containing the item, finds it using computer visual object detection, applies a learned grasping strategy to pick up the object, and finally navigates back to the user to deliver the item.

A video of this demonstration is available on the STAIR project web site:

`http://cs.stanford.edu/groups/stair`

Figure 4 shows the detailed organization of the components used in this demonstration. Computer 1 was an offboard Linux machine, Computer 2 was an onboard Windows machine, and Computer 3 was an onboard Linux machine.

(The number in parentheses after each process name indicates what computer it was run on, and the edges in the graph show the directions of TCP data flows.)

The processes used were subdivided roughly into five main partitions (also shown in the figure), which were implemented using Switchyard by about 4-5 largely disjoint (but collaborating) teams of researchers.

Spoken Dialog We now describe STAIR's spoken dialog system, specifically its implementation using Switchyard. (See also (Krsmanovic *et al.* 2006).) The audio_mic process continuously records audio, and streams it to sphinx_transcribe, which uses the CMU Sphinx speech-recognition system to continuously recognize speech. When the system recognizes a command, it passes the command to the central planner's fetch_item_director process. In the demonstration, STAIR's verbal acknowledgment of the command is then generated using the Festival speech-synthesis system, by the tts_festival process running onboard the robot.

Navigation A navigation system enabling STAIR to navigate in indoor environments and open doors is described in (Petrovskaya & Ng 2007). For this demonstration, we replaced portions of the software to increase its speed and robustness to changing environments. Our implementation used a Voronoi-based global planner and VFH+ (Ulrich & Borenstein 1998) to avoid local obstacles. As shown in Figure 4, the navigation and localization module used different processes to stream the laser readings, perform localization, generate low-level commands to control the Segway base, and so on. The processes with faster real-time requirements were generally run onboard the robot (computer 3); the processes running on longer time-scales, such as the Voronoi planner, were run on a more powerful offboard machine (computer 1).

Object detection (Gould *et al.* 2007) developed a foveal-peripheral visual object detection system that uses a steerable pan-tilt-zoom (PTZ) camera to obtain high resolution images of the object being recognized. Since object detection is significantly easier from high resolution images than from low resolution ones, this significantly improves the accuracy of the visual object detection algorithm. (Note that, in contrast, obtaining high resolution, zoomed-in images this way would not have been possible if we were performing object detection on images downloaded off the internet.) In our demonstration, a fast offboard machine (computer 1) was responsible for steering our robot's PTZ camera to obtain high resolution images of selected regions, and for running the object recognition algorithm. An onboard machine (computer 2) was used to run the low-level device drivers responsible for steering the camera and for taking/streaming images. Our object recognition system was built using image features described in (Serre, Wolf, & Poggio 2005).

Grasping To pick up the object, the robot used the grasping algorithm developed by (Saxena *et al.* 2006a; 2006b). The robot uses a stereo camera to acquire an image of the object to be grasped. Using the visual appearance of the object, a learned classifier then selects a good "grasp point"—i.e., a good 3d position at which to attempt to pick up the object. The algorithm for choosing a grasp point was trained on a large set of labeled natural and synthetic images of a variety of household objects. Although this training set did not include staplers, the learned feature set was robust enough to generalize to staplers. The low-level drivers for the camera and the robot arm were run onboard the robot (computer 2); the slower algorithm for finding a grasp point was run offboard (computer 1). An example of the robot executing a grasp of the stapler is shown in 5.

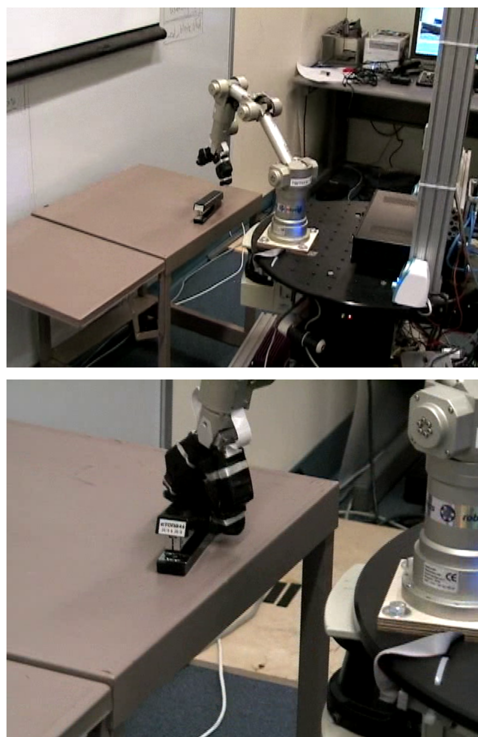

Figure 5: Robot grasping a stapler, using a learned grasping strategy.

Conclusion

In this paper, we described the hardware and software systems that allowed the STAIR robot to perform the "fetch a stapler" demonstration. The Switchyard software framework provided a uniform set of conventions for communications across processes, and allowed different research teams to write software in parallel for many different modules. Using Switchyard, these modules were then easy to execute simultaneously and in a distributed fashion across a small set of onboard and offboard computers.

Acknowledgments

The STAIR project is a large group effort involving a large team of researchers. For the "fetch an item" demonstration, the object recognition system was developed in collaboration with Steve Gould, Joakim Arfvidsson, Adrian Kaehler,

Ben Sapp, Marius Meissner, Gary Bradski, Paul Baumstarck, Sung Chung; the grasping system was developed in collaboration with Ashutosh Saxena, Justin Driemeyer, Justin Kearns, and Chioma Osondu; the spoken dialog system was developed in collaboration with Jeremy Hoffman, Filip Krsmanovic, Curtis Spencer, and Dan Jurafsky. We are also grateful to Quan Gan and Patrick Gallagher for their help building the STAIR 2 platform. Many others have contributed to the project's hardware and/or software, and are listed on the STAIR projects' web site, http://cs.stanford.edu/groups/stair. This work was supported by the NSF under grant number CNS-0551737.

References

Gerkey, B.; Vaughan, R.; and Howard, A. 2003. The Player/Stage Project: tools for multi-robot and distributed sensor systems. In *Internation Conference on Advanced Robotics (ICAR)*.

Gould, S.; Arfvidsson, J.; Kaehler, A.; Sapp, B.; Meissner, M.; Bradski, G.; Baumstarck, P.; Chung, S.; and Ng, A. Y. 2007. Peripheral-foveal vision for real-time object recognition and tracking in video. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*.

Krsmanovic, F.; Spencer, C.; Jurafsky, D.; and Ng, A. Y. 2006. Have we met? MDP based speaker ID for robust dialog. In *Ninth International Conference on Spoken Language Processing (InterSpeech-ICSLP)*.

Montemerlo, M.; Roy, N.; and Thrun, S. 2003. Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Petrovskaya, A., and Ng, A. 2007. Probabilistic mobile manipulation in dynamic environments, with application to opening doors. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Saxena, A.; Driemeyer, J.; Kearns, J.; and Ng, A. Y. 2006a. Robotic grasping of novel objects. In *Neural Information Processing Systems (NIPS)*.

Saxena, A.; Driemeyer, J.; Kearns, J.; Osondu, C.; and Ng, A. Y. 2006b. Learning to grasp novel objects using vision. In *International Symposium on Experimental Robotics (ISER)*.

Scholl, K.-U.; Albiez, J.; and Gassmann, B. 2001. MCA - an expandable modular controller architecture. In *3rd Real-Time Linux Workshop, Milan, Italy*.

Serre, T.; Wolf, L.; and Poggio, T. 2005. Object recognition with features inspired by visual cortex. In *IEEE Conference on Computer Vision and Pattern Recognition*.

Tira-Thompson, E. 2004. *Tekkotsu: A Rapid Development Framework for Robotics*. Master's Thesis, Carnegie Mellon University.

Ulrich, I., and Borenstein, J. 1998. VFH+: reliable obstacle avoidance for fast mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA)*.