Lewis, an Entry in the 2004 Robot Challenge

Michael Dixon and **William D. Smart**
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130
United States
{msd2,wds}@cse.wustl.edu

Abstract

This paper describes Lewis, the Washington University entry in the 2004 Robot Challenge event. We describe the new control architecture implemented for the 2004 competition, our high-level behavior-sequencing mechanism, and discuss our performance during the Challenge event.

Introduction

This paper describes the Washington University entry in the 2004 Robot Challenge event. The robot, Lewis, shown in figure 1 is an iRobot B21r mobile robot. It carries all the usual sensors, but relies primarily on the SICK laser range-finder, and a pair of stereo cameras on a pan-tilt mount.

Our entry this year builds on our successful system from the 2003 competition. however, almost all of the software was rewritten to incorporate lessons learned from last year. In particular, we have focused on fault-tolerance in our architecture, to enable hardware and software failures to be dealt with gracefully without interruption to the task being performed.

In this paper, we describe this fault-tolerant architecture, the task-sequencing system, and discuss our experiences at the competition. We conclude by offering some thoughts on the performance of our system, and where our research is heading now.

Our Approach

The AAAI Robot Challenge is a problem which requires the demonstration of a broad set of capabilities, and thus requires the development a set of general tools for operating in complex environments and interacting with people. Based on our experiences in the 2003 event, we decided to focus our efforts on designing a modular control architecture, with built-in fault tolerant features. The modular system allows great flexibility on-site, since pre-coded modules can be combined using a simple scripting language, reducing the need for (possibly error-prone) on-site programming. The emphasis on fault-tolerance acknowledges that failures will happen, and that the best we can do is to reduce their severity and duration when they do.

Figure 1: Our robot, Lewis.

Our aim was to improve the manageability and reliability of our system by compartmentalizing functionality into modular services that could be combined to create greater complexity, but could be individually debugged and tested with greater ease than an integrated system. This would also improve the system's flexibility, giving us the ability to go into a previously unknown environment and assemble existing modules to perform a specific set of tasks in a short amount of time. To that end, our goal was to develop a useful set of general-purpose components, and to solve the challenge tasks using these pieces. To sequence these sub-tasks, we developed a simple FSM-based system, controlled using a scripting language.

Because of the complexity of robot systems, and the unpredictability of real-world environments, fault tolerance is another important concern. For instance, a fail-

ure in the laser rangefinder hardware or software could result in a robot blindly colliding with an obstacle, resulting in possible damage or injury. Thus the system must have the capability to automatically detect failures of components and respond appropriately. One such response would be to completely shut down the system, preventing risk of damage; however, in many cases, redundant components may make it possible to cope more gracefully with failure. For instance, if one of the system's two cameras fails, the most desirable response is to use the second camera as a back-up and to continue functioning instead of simply shutting down. Thus a focus of our approach was to design redundant components for our critical systems and develop the architecture to support graceful degradation of system performance in the event of failure.

The Control Framework

Our goals of modularity and fault tolerance drove the development of a new framework for creating and managing the components of our system. This framework allows us to define a set of generalized, abstract interfaces, implemented by service processes. These services are managed by a central control processes known as the MCP.

Interfaces

Interfaces define an object-like functional unit. They are defined by an XML-based format which specifies the data a given unit makes available and the commands it responds to (as shown in figure 2). The interface itself is purely abstract and is independent of any implementation. These interfaces define the data that applications can work with. For example, range sensors might provide a `Distance` interface (in addition to their raw readings). As we will show below, these abstractions allow us to provide a level of fault-tolerance not possible if we deal directly with the output of any particular device.

Services

A service is a separate process that contains the implementation of one or more interfaces. Services are written in C++ and extend from a parent Service class. Though services are a stand-alone process, they make data available to other processes by publishing it in blocks of shared memory and receive messages via command pipes. The details of this inter-process communication are abstracted from the user by a client class which allows a user to interact with a service via an object-like interface. These IPC mechanisms give us a very low-overhead communication between processes, allowing us to have strong encapsulation at almost no additional cost.

As an example, a camera service implements an `ImageSource` interface for accessing the image the camera sees and a `CameraControl` interface for controlling zoom and exposure. An ImageSource client that connected to this service would be able to read the image

```
<?xml version="1.0"?>
<!DOCTYPE interface SYSTEM "interface.dtd">
<interface name="RangeFinder" version="1.0">
  <description>
    Generic range-finder for laser, sonar, etc.
  </description>
  <types filename="types.xml"/>

  <comment>
    If uniform is true, the offsets and
    separations can be used by client code
    to easily iterate through each element
    of the range sensor (e.g. laser).  For
    more exotic devices (e.g. sonar), set
    uniform to false, and publish a Pose3D
    for each element.
  </comment>

  <publish>
    <data name="range_count" type="Integer"/>
    <data name="ranges" type="Float"
      array_size="range_count"/>
    <data name="max_range" type="Float"/>
    <data name="uniform" type="Boolean"/>
    <data name="sensor_positions" type="Pose3D"
      array_size="range_count"/>
    <data name="x_offset" type="Float"/>
    <data name="y_offset" type="Float"/>
    <data name="z_offset" type="Float"/>
    <data name="theta_offset" type="Float"/>
    <data name="phi_offset" type="Float"/>
    <data name="psi_offset" type="Float"/>
    <data name="x_separation" type="Float"/>
    <data name="y_separation" type="Float"/>
    <data name="z_separation" type="Float"/>
    <data name="theta_separation" type="Float"/>
    <data name="phi_separation" type="Float"/>
    <data name="psi_separation" type="Float"/>
  </publish>
</interface>
```

Figure 2: The XML interface description language.

data from shared memory via a call to `get_image()`, and a CameraControl client connected to this service could pass command messages to the service via functions such as `set_zoom()`.

The Service Broker

The service broker keeps track of all the interfaces available, and which services implement them. When an application requests a service, this request goes through the broker, which chooses the most appropriate service to provide the interface. The requester and service are then connected directly. This function of the broker is very similar to that performed by a CORBA ORB.

The broker allows us to centralize policy for service selection, and isolates the application from the need to know where services are actually implemented. It also allows us to perform optimizations by cleverly selecting services to provide the requested interfaces. Any particular application will request a set of interfaces which will, in general, be satisfiable using several sets of services. The broker can select services to optimize a number of criteria (quality, data rate, computational cost, *etc.*). The system deployed at the 2004 competition did not perform such an optimization, but we are currently working on this feature.

In addition to service selection, the broker monitors services to detect unexpected termination. In the event of such a failure, the broker attempts to restart the service and reconnect its clients. If the service repeatedly fails, or identifies itself as being out-of-service (see below), the broker will select another service to provide the requested interfaces, and automatically reconnect all affected clients. This gives a measure of fault-tolerance to the system, which we discuss in more depth below.

Example Services

Our implementation involved a large network of services ranging from low-level components for interfacing with hardware, like the laser range-finder, to higher-level components, such as sign detection. This section describes some of the services used in our system.

`RFlexBase` The RFlex hardware provides access to our robot's motor velocities, odometry, sonar, and power level. The `RFlexBase` service communicates with this hardware over the serial port, implementing the `Synchrodrive`, `Egomotion`, `RangeFinder`, and `Power` interfaces.

`SICKLaser` This service continually reads the data provided by the SICK PLS laser rangefinder over the serial port and makes it available via the `RangeFinder` interface.

`VectorMover` `VectorMover` smoothly moves the robot directly to a given goal point. It computes the velocities needed to move the robot to the goal and sends the appropriate commands to its `Synchrodrive` client. While still a very low level interface, it exposes a set of functions that allow application developers to rotate or translate the robot specific amounts without worrying about wheel velocities.

`ObstacleAvoider` `ObstacleAvoider` implements the same interface as `VectorMover`, but uses a `RangeFinder`, such as the laser, to detect the presence of obstacles and chooses a path which avoids them. It uses a greedy strategy, continually computing the set of all points accessible via straight-line paths and approaching the point from this set which is closest to the goal. Rather than sending its commands through the `Synchrodrive`, it moves using the higher-level interface provided by `VectorMover`.

`IEEE1394Camera` This service is another hardware interface, which communicates with all cameras on the Firewire bus and makes available their image data and allows control of zoom, focus, iris, shutter, and gain.

`BlobFinder` This service takes in data from the camera service and uses an adaptation of the CMVision blob detection algorithm to find contiguous regions of certain colors in the image and publishes their sizes and positions.

`BlobFinder3D` This service takes the output of `BlobFinder` and attempts to use the laser to measure the distance to each. Given this distance, it computes and publishes the 3D position and absolute size of each blob.

`SignDetector` `SignDetector` is used to look for specific signs placed to guide the robot through unknown environments. Since the signs it is trying to find are white with black arrows, it uses `BlobFinder3D` to find white and black blobs and searches for white blobs with black blobs nested inside them. When nested blobs of the right size and position are found, it tests the original image to measure the arrow direction, and publishes the 3D coordinates of the sign and the direction of its arrow.

The Sequencer

In the spirit of modularity, we organize our applications as sequences of independent task-achieving modules. For example, the first part of the challenge event, to get from the door of the conference center to the registration desk would be handled by sub-task modules for navigating doorways, finding and following signs, navigating elevators, standing in line at the desk, and interacting with the registrar. Each of these sub-task applications must be sequenced correctly for the overall application to succeed. We accomplish this through the use of a simple finite state machine-based sequencer, controlled by a scripting language.

An example of the scripting language is shown in figure 3. Each sub-task corresponds to a state in the FSM, with the transitions between states controlled by the exit codes of these applications. The script defines symbolic names for each state, along with the executable for the application. It also enumerates all of the expected

```
start door
restart 3
recovery oops
use gui

state door /usr/bin/DoorNavigator
  action 0 trans sign

state sign /usr/bin/SignFollower
  action 0 trans sign
  action 1 trans queue

state queue /usr/bin/QueueStander
  action 0 trans register

state register /usr/bin/Register
  action 0 trans stop

state oops /usr/bin/Recovery
  action 0 trans stop
```

Figure 3: A sample sequencer script.

return codes, and which transitions should be made on them. The start state is also given (`door`).

The sequencer monitors these sub-task applications for failure. If a process terminates unexpectedly, it will be restarted up to a limit (specified by the argument to `restart` in the script). If this limit is exceeded, or if an unexpected return code is encountered, a special recovery state is entered. This allows the robot to enter a safe, known state if something goes wrong. For the competition, this was a state where the robot looked for and followed a red baseball cap. Once the cap vanishes from sight, a graphical interface is shown on the robot's touch screen (specified by `use gui` in the script), allowing the human supervisor to select the state that the robot should start off from.

In the example script shown in figure 3, the robot starts out in the `door` state, transitioning to the `sign` state once it is through. This state looks for a directional sign, drives to it, and follows it. It terminates under two conditions. If another directional sign is seen, it terminates and returns 0. If the registration desk is seen, it terminates with a 1.

Although this sequencing system is quite simple, we have found it to be sufficient for the challenge tasks. It's main advantage, however, is that it allows us to sequence together pre-written applications easily, with recompilation. This greatly reduces the amount of time needed to assemble and deploy an application while at the event. The inherent modularity also allows us to develop the sub-tasks independently, as long as the pre- and post-conditions are well specified.

Fault-Tolerance

The fault tolerance in the system is implemented in a number of ways.

Strong Modularity

The entire framework is organized in a strongly modular fashion. In particular, most elements of the framework run as seperate processes under the linux operating system. This gives us protection from software errors and unexpected terminations. If something goes wrong with a piece of code, the fault is isolated to the module that it occurs in. This modularity also makes finding and debugging problems easier, since each module is typically a small amount of code.

Sequencer Monitoring

Each of the sub-tasks controlled by the sequencer is monitored for unexpected termination or return codes. Since it is impossible to write completely error-free code, this allows us to deal with software problems when they occur. The main advantage of this is that it lets us continue with the deployment, keeping the robot operating, in the face of occasional software failures.

In the case of failures that the sequencer cannot deal with, such as repeated abnormal terminations, a human supervisor is asked for help. This is a simple form of shared-initiative control, where the robot asks for help when it deems it to be needed, rather than needing the supervisor to constantly monitor the system.

Service Broker Monitoring

Similarly to the sequencer, the service broker monitors the status of services, and can restart them when needed. More importantly, however, is the ability to substitute in working services for failed ones. For example, `RangeFinder` is a generic interface for `LaserRangeFinder`, `SonarRangeFinder`, and `StereoRangeFinder`. By default, an application program which asks for a `RangeFinder` will be connected to the `LaserRangeFinder` because of its greater accuracy. But if the laser malfunctions, the service broker can recognize this and switch the application program over to `SonarRangeFinder`, without any explicit action by the application. Although this will result in different behavior, since a sonar is not the same as a laser, it should provide a graceful degradation in the face of failures. If we depend on the laser for input, as many current applications do, we are tied to that device. If it fails, the application fails. However, if we are willing to work with an abstraction, we can gain some robustness.

Deployment Experiences

Our performance this year was not as good as last year (at IJCAI in Acapulco, Mexico). We completely rewrote the control framework in the intervening year, and there were some bugs that had not been completely worked out by the time of the competition. More of a problem, however, was our lack of a library of robust services. Our intention was to compose a solution to the challenge task using pre-build services, sequenced by the sequencing script. However, we found that our

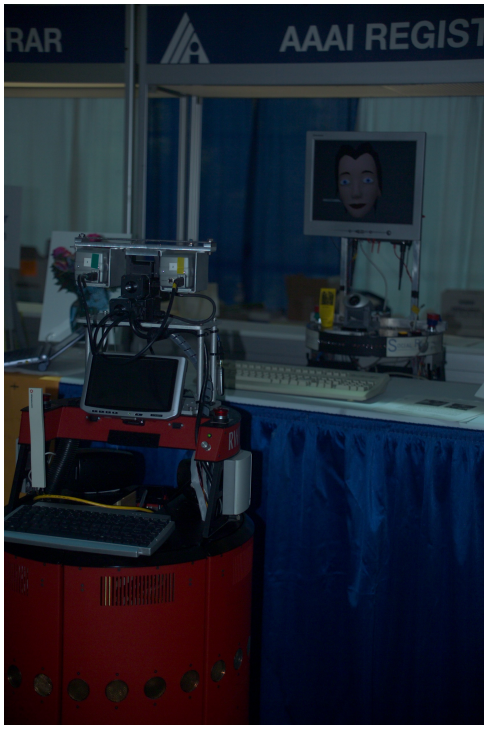

Figure 4: Lewis talks to Grace.

existing services were not adequate for the task, and we had to rewite several of them on-site, under time pressure. This led to a poor performance on the overall challenge task.

However, we did successfully demonstrate the fault-tolerance of the framework in a demonstration during the robot exhibition. While performing a simple obstacle-avoidance task, the laser device was disabled. The failure was recognized by the system, and the service broker started up the sonar sensors to provide the `Distance` interface to the application. This allowed the application to continue seamlessly under the failure of the laser.

During the event, we also interacted with GRACE, acting as a robotic receptionist (see figure 4). Although much of the interaction was performed using the wireless network, synchronized speech output was added to give the illusion of human-like communication. While not the most technical of our achievements, it was one of the most crowd-pleasing moments of the competition.

Conclusions and Current Work

Although our control framework performed well, our overall performance was less compelling. We attribute this to our lack of well-tested services at the time of the event. Our experience confirmed what we already knew: It is not possible to implement a working solution in the field. Such a solution must be composed of pre-written and pre-tested modules, sequenced together appropriately. In preparation for next year, we have begun implementing a wider variety of services on which to base our entry.