DDX: A Distributed Software Architecture for Robotic Systems

Peter Corke, Pavan Sikka, Jonathan Roberts and Elliot Duff

CSIRO ICT Centre

PO Box 883, Kenmore, Qld 4069, AUSTRALIA

Email: firstname.lastname@csiro.au

Abstract

The Dynamic Data eXchange (DDX) is our third generation platform for building distributed robot controllers. DDX allows a coalition of programs to share data at run-time through an efficient shared memory mechanism managed by a *store*. Further, *store*s on multiple machines can be linked by means of a global *catalog* and data is moved between the *store*s on an as needed basis by multi-casting. Heterogeneous computer systems are handled. We describe the architecture of DDX and the standard clients we have developed that let us rapidly build complex control systems with minimal coding.

1 Introduction

For nearly 10 years we have been building complex robotic systems. Over time we have developed a series of software platforms to facilitate this endeavour. This paper describes our third generation platform which we call DDX for Dynamic Data eXchange. It builds upon ideas we have previously presented in Roberts et. al. [Roberts *et al.*, 1999]. Analyzing the common requirements from many applications we find the following issues to be important:

- an application is composed from numerous small, simple and well-tested programs that form a run-time coalition;

- an application may involve several processes running on a heterogeneous network of computers;

- the need for a low-overhead, but safe, mechanism to share data;

- an ability to log data to disk for debugging and presentation; and

- associating timestamps with all data.

There is a considerable literature on the topic of robot software architectures. The IPC [Simmons and James, 2001] system, as well as systems based on CORBA [Schmidt, 2004], provide similar functionality. They both provide the means to transfer data objects and synchronize between processes running on heterogeneous computer systems. They are both based on underlying network communications. A significant disadvantage of these systems is that even if all processes run on the one computer the data will pass through a significant portion of the network protocol stack. All communication in these systems is point-to-point; therefore, there is no single place at which all communications can be monitored. Furthermore, this causes data duplication when there are multiple clients.

IPC has bindings for C, C++, Allegro Common Lisp and Java, and has been ported to Solaris, Linux, WinNT, Win98, VxWorks, IRIX and MacOS. The `central` server provides a switch-yard for messages (by default) and can provide logging of message traffic. The TAO CORBA package[Schmidt, 2004] has C++ bindings and the object request broker (ORB), called TAO, has been ported to most versions of Windows (include WinCE) and Unix, VMS, LynxOS, VxWorks, QNX Neutrino, and OS9. IPC and TAO are both available in source form with generous license conditions. An application of TAO to tele-robotics is given in [Bottazzi *et al.*, 2002].

CORBA-based approaches require a description of the data objects, an IDL file, that must be compiled to provide access to the object. In contrast, IPC and DDX both use run-time type descriptions. This has a significant advantage in terms of simplifying code development.

An alternative form of communication that has become popular recently is publish/subscribe [th Eugster *et al.*, 2003; Pardo-Castellote *et al.*, 2001; 1997]. This form of communication has also been recently standardized by the OMG. This form of communication has been used in industrial networks such as DeviceNet but is only now beginning to find some use in robotics applications. Unlike the communication systems mentioned above, this form of communication optimizes network

TCP/IP Connection
Shared Memory Create and Manage
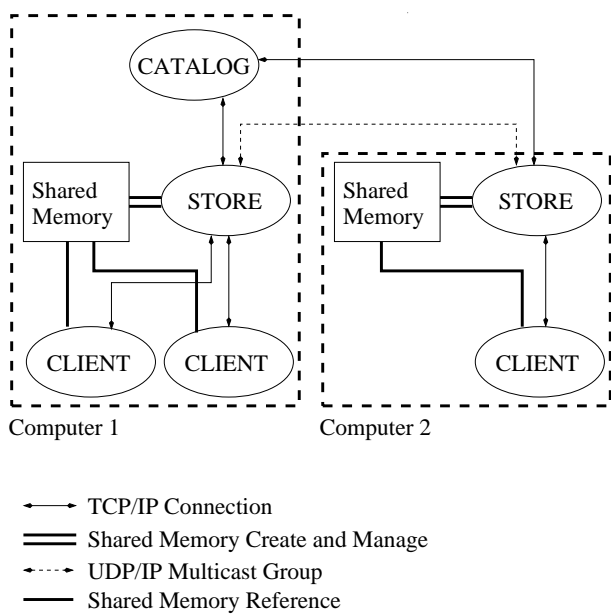UDP/IP Multicast Group
Shared Memory Reference

Figure 1: DDX components.

traffic amongst various computers so that a data packet is transmitted only once and is then delivered to all clients across the network only once. It maps onto the mechanism of multi-casting as defined by the UDP/IP protocol. DDX relies on this form of communication to transfer data between processes running on networked computers. DDX also provides a very efficient shared memory mechanism to allow processes running on the same computer to share data.

The remainder of this paper is structured as follows. Section 2 describes the fundamental mechanisms for data exchange, and Section 4 describes a large family of compliant clients that can be assembled into an application. Section 5 describes the means to coordinate the execution of the components across multiple computers, and Section 7 concludes and presents current directions.

2 Core components

DDX provides its functionality through two programs (`catalog` and `store`) and a client library `libddx.a`. Figure 1 shows how the various components can be put together to implement a distributed system based on DDX.

The `store` implements most of the functionality required for a single machine. The `store` provides client applications with a block of shared memory that is used by clients to store shared data. The store also provides clients with global process-shared semaphores (in shared memory) that are used for synchronized access to the shared data.

The `catalog` provides a global repository of data stored across machines in a distributed system. DDX requires a `store` to be running on each machine that forms part of the application. All of these distributed `store`s reference one `catalog` that is common to the application. The `catalog` controls the transfer of data amongst the stores by instructing stores to transmit data updates to all other stores using UDP/IP multi-cast.

2.1 Store

The `store` is one of the key components of DDX. It provides its clients with the ability to create and share data efficiently. It also allows synchronized access to the shared data. Finally, it allows clients to specify shared data as native 'C' data-types. This ability is critical to multi-architecture support.

Shared memory

The `store` creates and manages a block of shared memory used by clients to store data. The `store` maintains an internal table that contains information about the data stored in its shared memory. Clients map this shared memory into their own process-space and then have direct access to the data through memory pointers.

The `store` "understands" native 'C' declarations (including typedefs). When a client wishes to access some data in the `store`, it provides the `store` with a 'C' declaration for the data. The client is responsible to provide all type definitions required to parse the data declaration.

The `store` parses the data declaration provided and then allocates space in the shared memory block for the data. The `store` respects the architecture constraints in terms of primitive data sizes and alignments. The `store` then provides the client with the location of the data in the shared memory block.

This ability to understand native 'C' data declarations is key to efficient storage and manipulation of shared data. On a single machine, the shared data is stored in the native format. Therefore, clients are able to access data as if it was local to the program. The representation is efficient since no data translation is required (cf other RPC mechanisms). Clients use native 'C' declarations to define the data to be stored in shared memory. The only additional requirement is that the client needs to generate a string representation of the data declarations so that they can be communicated to the `store`. This can easily be done using some standard macro facilities provided by 'C'.

Global process-shared semaphores

The store creates a block of shared memory for global process-shared semaphores. The store creates a semaphore for each client for each data-item referenced by the client. These semaphores allow the clients to have synchronized access to the data.

Implementation issues

The `store` is implemented in 'C'. It confirms to the POSIX.1, POSIX.1b and POSIX.1c standards, and this is both an advantage and a limiting factor. Since the implementation makes extensive use of the facilities provided by these standards, DDX is only available on systems that provide fairly complete implementations of these standards.

We have tested the implementation on Solaris (Sparc), Linux (i586+), LynxOS (i586+) and QNX Neutrino (i586+). The Linux implementation requires the NPTL library and the 2.6 kernel (or a 2.4 kernel with Redhat patches to support NPTL).

2.2 Catalog

The `catalog` is critical to the sharing of data across computers in a distributed DDX application. It provides a global repository of information across all the computers that make up a distributed DDX system. The `catalog` maintains a list of all the `store`s connected to it. The `catalog` also maintains a list of all the data-items that have been created by clients. Each element of this list further contains a list of the `store`s that have clients with references to the data-item.

Whenever a client requests access to some data, the `store` checks to see if the data exists locally. If not, the `store` forwards the request to the `catalog`. If the data does not exist in the system, the catalog records information about the data in its internal tables and then instructs the `store` to create the data.

Inter-store communications

However, if the data already exists, the implication is that it was created by a client on some other `store` within the application. Therefore, the data needs to be shared across `store`s. The `catalog` accomplishes this by:

1. instructing the `store` to create the data and to enable multi-cast, and

2. instructing the other `store` to enable multi-cast for this data-item.

This sequence of instructions causes the `store`s to create internal threads that "listen" for multi-casts of the data-item and then update the data-item whenever a successful multi-cast message containing the data-item is received. These threads also provide any translation that may be required due to differences in architecture (size, alignment and byte-order). The multi-cast messages are posted automatically whenever the data-item is updated by a client.

When the clients are finished accessing the data, they inform the `store` which then informs the `catalog`. When the number of stores that have clients interested in the data-item comes down to 1, the `catalog` instructs the `store` to disable multi-cast for the data-item.

3 Client API

The following short example of C code shows the major features of the DDX client API. The code reads the value of an encoder from the store. Note that another program (the producer) is responsible for actually interfacing with the encoder and writing its value to the store. Often, a DDX client is both a "producer" and a "consumer".

In summary, the code performs the following steps:

1. Declare a structure using the `DDX_STORE_TYPE` macro. Typically this declaration is performed in a header file that is shared by all clients with an interest in the structure.

2. The `ddx_client_init()` function is called which initializes the shared memory data structures.

3. The `ddx_store_open()` function is called that opens a connection to the store running on the local machine. This function returns a handle to the store.

4. The encoder structure is looked-up in the store using the `ddx_store_lookup_item()` function. This function returns a handle to the encoder structure.

5. The value of the encoder structure is read from the store continuously while its data is valid. The data is read using the `ddx_store_read()` function which can be used in a blocking or non-blocking mode. In this example the read is blocking and will only return when new data arrives in the store.

6. Finally, the store client side is cleaned up.

```
/* Declare encoder type */
DDX_STORE_TYPE (Encoder,
  struct {
    double value;
    int    valid;
  } );

/* Declare variables */
DDX_STORE_ID    *storeId;
DDX_STORE_ITEM  *encoderItem;
Encoder         enc;

/* Initialise DDX client side */
ddx_client_init(0);

/* Open local store */
storeId = ddx_store_open(NULL, 0, 2);

/* Lookup encoder structure */
encoderItem = ddx_store_lookup_item(
    storeId, "encoder", "Encoder",
    sizeof(Encoder));

/* Print out value */
enc.valid = 1;
while (enc.valid) {
```

```
    ddx_store_read(encoderItem, &enc,
      NULL, 1.0, 1);
    printf("val = %f\n", enc.value);
}

/* Cleanup DDX client side */
ddx_store_done_item(encoderItem);
ddx_store_close(storeId);
ddx_client_done();
```

4 Standard clients

4.1 Sensors

We have developed several applications that acquire data from various standard robotic sensors and write it to the store:

gps A generic GPS utility that reads standard NMEA messages from a serial port and writes them as structures to the store.

imu A generic IMU utility that reads data from a Crossbow DMU or CSIRO EiMU IMU via a serial port and writes the data to the store.

pls A utility that interfaces with a SICK PLS or LMS 2D laser scanner via a serial port and writes data to the store.

IIO IIO is a library developed by the Automation group to streamline access to sensors. It was originally developed for Industry Pack (IP) modules within a VME/VxWorks environment but has since been expanded to include support for the PC parallel port and Modbus/TCP modules within a POSIX framework. It provides a uniform API to access a large variety of sensors that are easily described in a configuration file. The DDX IIO application reads its configuration from a file. It allows for different sample rates for individual sensors. It reads sensor data and writes it to the store. It can also read data from the store and write it to output modules (digital/analog outputs).

Command line options on these utilities allow us to specify the variable name within the store, update rate and other sensor specific options. These options can also be specified in a configuration file.

4.2 Logger

In our experience with commissioning many robotic systems, we found that lack of data about what happened prior to a failure was a major limitation. With the DDX architecture all data pertinent to an application passes through the store and this is an ideal point at which to record it. The logger application accepts command line options (or a configuration file) specifying the variables to be logged and the interval (can specify every nth update to be recorded, useful for high frequency signals). A separate thread within the logger monitors each variable

and changes are written to a ring buffer. The priority of each thread can be specified if required. The ring buffer is written to disk by a low-priority thread within the logger. Log files can be rolled over when they reach a specified size, after a specified time, or on receipt of a Unix signal.

The logger files have a hybrid structure. An ASCII header contains the start time and date, user name and the data organization for the binary portion of the file (endian-ness, alignment). It then describes the types of all logged variables in a 'C' like syntax. Following this is the data, each object is written in native binary format with a time stamp and record type. The files are therefore compact but completely self-contained, since the definition of the data-types is embedded in the file, giving robustness with respect to changes in data types.

Log file reading
The `ddxreadlog` utility allows a user to parse a log file, and automatically uncompresses the file if it has been compressed. Options provide the ability to list the variables within the file, the number of objects of each type within the file, or the numerical values of specified variables. The variables can be dumped in ASCII tabular format (compound objects are "flattened") or as a Matlab MAT-file in which case compound object structure is maintained. An example session is shown below.

```
ratbert.cat.csiro.au% ddxreadlog -l 20040312153345.log
3 top level variables in file
tr_pos [860]
tr_demand [602]
ddx_gps [602]
```

Matlab interface
A MATLAB M-file provides a convenient wrapper to `ddxreadlog` and can optionally re-sample all extracted variables to the timestamps of the highest sample rate variable extracted. Having the logged data available within Matlab greatly simplifies analysis, debugging and data presentation.

For example, the datafile from above can be loaded and assigned to the workspace object **g** whose elements are vectors of signals whose corresponding time value is given by the element **t**. The following example illustrates this use of ddxreadlog.

```
>> ddxretrieve('20040312153345.log', ...
        'g=ddx_gps')
 Fastest sampling is ddx_gps at 0.100000s
 Assign ddx_gps -> g
>> g
g =

          t: [602x1 double]
    Easting: [602x1 double]
   Northing: [602x1 double]
    velEast: [602x1 double]
   velNorth: [602x1 double]
    absDist: [602x1 double]
     absVel: [602x1 double]
    initEast: [602x1 double]
```
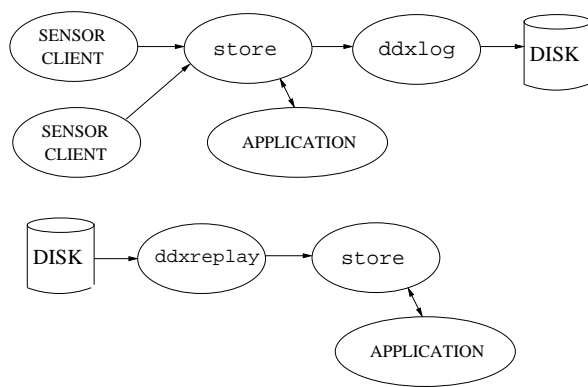
Figure 2: Top: the 'live' mode of operation where data is logged to disk. Bottom: the off-line situation where data can be replayed through the store.

```
    initNorth: [602x1 double]
   gpsStatus: [602x1 double]
       x_dmd: [602x1 double]
       y_dmd: [602x1 double]
     vel_dmd: [602x1 double]
   steer_dmd: [602x1 double]
     heading: [602x1 double]
>>
```

Logger API

The hybrid logger data files have become our defacto standard for recording data. We find it useful in some applications to read or write these structured files directly from an application, without the need to run a store and catalog. A simple 'C' language API provides this functionality.

4.3 Replayer

Most control applications take a stream of sensor values and compute an appropriate control demand. At the same time, data is logged to disk for off-line analysis. Figure 2(top) shows this mode of operation where data is logged from the store to disk (using `ddxlog`). To facilitate testing and debugging offline, we can replay logged sensor data through the store using the `ddxreplay` program. Figure 2(bottom) shows this mode of operation.

The data can be played back at the same rate as it was originally generated. This results in a more realistic playback to the rest of the system. During a replay session, the sensor producer clients (the programs that actually interface to the sensor hardware) are not run. Instead the `ddxreplay` program is run with a log file containing all the sensor data. This mechanism is an extremely powerful way to aid with debugging of application code, as this code can not tell the difference between real 'live' data and replayed 'stored' data.

4.4 Interactive text console

The facility to examine the status of a store variable through a command line interface is extremely useful. The `ddxsh` utility allows us to list all registered values and their types, check or change the current value, and to see the last time the value was updated. An example session is shown below:

```
ratbert.cat.csiro.au% ddxsh -s emma-ph
store@emma-ph: ls
cmplx
pls
test
store@emma-ph: ls -l
cmplx   [size = 3168]   [count = 1133]
pls     [size = 1028]   [count = 1133]
test    [size = 8]      [count = 0]
store@emma-ph: ls pls
struct {
        int     numPoints;
        int     range[256];
};
store@emma-ph: ls test
struct {
        int     x;
        int     y;
};
store@emma-ph: print test
0.0 [0]
struct {
        int     x       = 0;
        int     y       = 0;
} test;
store@emma-ph: set test 1.0 2.0
OK
store@emma-ph: print test
1079323091.432622000 [1]
struct {
        int     x       = 1;
        int     y       = 2;
} test;
store@emma-ph: ls -l
cmplx   [size = 3168]   [count = 1199]
pls     [size = 1028]   [count = 1199]
test    [size = 8]      [count = 1]
store@emma-ph: quit
ratbert.cat.csiro.au%
```

4.5 Signal generator

`ddxsiggen` is a utility that can generate different waveforms (for example, sine, cosine, triangle, square etc) and write them into the `store`. The waveforms to be generated and their properties are read from a configuration file. An example configuration file is shown below:

```
%
%  Comment (Example configuration file)
%

interval 0.01

signal sine
type sine
```
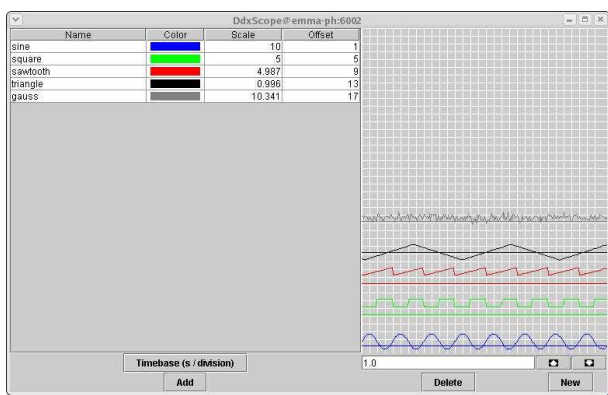
Figure 3: A screen-dump of the `ddxscope` utility.

```
params
  A = 10   % Amplitude 10
  C = 0    % Phase angle 0 rad
  f = 0.5  % Frequency 0.5 Hz
  D = 5    % Offset 5

signal square
type square
params
  A = 5
  B = 10
  T = 2
```

4.6 Scope

`ddxscope` is a GUI written in Java. It allows the user to select individual items from the store and then displays them on a virtual oscilloscope. Figure 3 shows a screen-dump of this utility. The signals were generated using the `ddxsiggen` utility described above.

4.7 Embedded microprocessor interface

Our embedded systems, based on HC12 and Atmel processors with a custom multi-threading kernel have the ability to "export" variables used by the program. A serial line protocol enables these variables to be read (polled or periodic) or written. A proxy client reflects these variables into the store. Therefore when a store variable is written, the client sends a command over the serial port to modify that variable in the memory of the embedded system. Likewise, values can be read periodically and updated in the store. This facility allows ready integration of low-cost embedded systems within the overall control architecture.

5 Launcher

As described above, we use DDX to assemble at run time a coalition of programs to share data and perform a control function. In general most of the programs in the coalition are standard: store, logger, and sensor clients. Generally only the control program itself varies from ap-

plication to application, and even then we are often able to recycle a considerable amount of code.

The members of the coalition may need to run on different computers in the network, and starting and stopping the coalition became something of a problem. Our answer to this is a helper application we call the `launcher`. A configuration file specifies a number of operational run levels (somewhat analogous to the Unix `init.d` mechanism) and the programs required at each level. Typical run levels are:

0 No programs running.

1 Catalog started

2 Stores started on all participating machines.

3 Sensor clients started on all participating machines.

4 Application specific, perhaps low-level loops

5 Application specific, perhaps task planner

6 etc.

The launcher program can take a command line option to run the system up to a specified level, or in interactive mode the level can be changed at will with the programs started or terminated appropriately. If any program in the "stack" fails (detected by signals) the level is dropped to the level below that which the failed program ran at.

Each line in the launcher configuration file contains the launch level, name of the program, the host it is to run on and command line options. An additional option indicates whether the program is "launcher aware", i.e., it uses a launcher API to signal its launcher that it has initialized and that the launch sequence should proceed. We originally used the `rsh` mechanism but found it was not sufficiently portable or reliable across platforms.

An example configuration file is shown below:

```
% Curb following launch file
#undef linux
1 catalog launch-aware 10 catalog
2 store launch-aware 10 store
3 pls launch-aware 10 pls -d ttyS8 -S
4 gps launch-aware 10 gpslog -store
    -nmea ttyS7:9600 -message gpgga,gpllk
    -type sr530
5 tractor-server launch-aware 10
    tractor-server
6 curb launch-aware 10 curb
7 snake launch-aware 10 snake curbLaser
8 tractor-test launch-aware 10
    ../bin/i486-linux/tractor-test
```

6 Future Work

One of the key benefits of DDX is that it provides a useful level of device abstraction to the consumer. Since the consumer does not need to be aware of how data from a device is created by the producer, the data could be real, pre-recorded, virtual or simulated.

This feature was demonstrated in the control of an autonomous tractor [Usher *et al.*, 2004], where DDX was used to create a virtual PLS that enabled the re-use of existing wall-following code. In this case, the virtual walls were created from the segmentation of the road from an omni-directional video camera.

The value of such device abstraction has been well documented. Of particular note is the Player/Stage platform[Vaughan *et al.*, 2003]. In future work, we intend to write a number of Player device drivers (laser and position) that communicate with the store. We have also recognized that Player/Stage could benefit significantly by using DDX as an additional transport layer.

Other future work will include language bindings for Java, Python and Matlab. The latter will allow functionality like Real-Time workshop where Simulink blocks can directly read and write objects in the store, measuring real-world values and performing control.

7   Conclusion

We have described a distributed software architecture for robot control systems that is currently deployed in nearly 10 systems. These span the range from massive excavators with several networked control computers, to embedded systems, to high performance servo systems with 1 kHz sample rates.

This architecture allows us to rapidly develop prototype robotic systems with the following attributes:

- an application is composed from numerous small, simple and well-tested programs that form a run-time coalition;

- an application may involve several processes running on a heterogeneous network of computers;

- the architecture provides application processes with a low-overhead, but safe, mechanism to share data;

- an application has a built-in ability to log data to disk for debugging and presentation.

Acknowledgment

References

[Bottazzi *et al.*, 2002] S. Bottazzi, S. Caselli, M. Reggiani, and M Amoretti. A software framework based on real-time CORBA for telerobotic systems. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pages 3011–3017, Lausanne, October 2002.

[Pardo-Castellote *et al.*, 1997] Gerardo Pardo-Castellote, Stan Schneider, and Mark Hamilton. Ndds: The real-time publish-subscribe middleware. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1997.

[Pardo-Castellote *et al.*, 2001] Gerardo Pardo-Castellote, Stefaan Sonck Thiebaut, Mark Hamilton, and Henry Choi. Real-time publish-subscribe protocol for ip-based real-time communication. *Instrument Society of America*, 2001.

[Roberts *et al.*, 1999] J.M. Roberts, P.I. Corke, R.J. Kirkham, F. Pennerath, and G.J. Winstanley. A real-time software architecture for robotics and automation. In *Proceedings of IEEE Int. Conf. on Robotics and Automation*, pages 1158–1163, Detroit, USA, 1999.

[Schmidt, 2004] Douglas C. Schmidt. Real-time CORBA with TAO, 2004.

[Simmons and James, 2001] Reid Simmons and Dale James. *Inter-Process Communication*. Carnegie-Mellon Univeristy, 3.4 edition, February 2001.

[th Eugster *et al.*, 2003] Patrick th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[Usher *et al.*, 2004] Kane Usher, Jonathan Roberts, Elliot Duff, Peter Corke, and Graeme Winstanley. Road following using virtual range sensing and radially constrained active contours. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004. Submitted.

[Vaughan *et al.*, 2003] R. T. Vaughan, B P Gerkey, and A Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2421–2427, Las Vegas, USA, 2003.