A framework for robust mobile robot systems

Nik A. Melchior and William D. Smart

Washington University in St. Louis, One Brookings Drive, St. Louis, MO, USA 63130

ABSTRACT

Fielded mobile robot systems will inevitably suffer hardware and software failures. Failures in a single subsystem can often disable the entire robot, especially if the controlling application does not consider such failures. Often simple measures, such as a software restart or the use of a secondary sensor, can solve the problem. However, these fixes must generally be applied by a human expert, who might not be present in the field. In this paper, we describe a recovery-oriented framework for mobile robot applications which addresses this problem in two ways. First, fault isolation automatically provides graceful degradation of the overall system as individual software and hardware components fail. In addition, subsystems are monitored for known failure modes or aberrant behavior. The framework responds to detected or immanent failures by restarting or replacing the suspect component in a manner transparent to the application programmer and the robot's operator.

Keywords: autonomic computing, recovery–oriented computing, fault tolerance, mobile robotics, software architecture

1. INTRODUCTION

Mobile robotics relies heavily on the integration of components into a coherent system. Robots carry hardware sensors and actuators produced in a market less commoditized than that of the personal computer, and application program interfaces (APIs) and device behaviors are not yet standardized. Although certain algorithms are becoming standard for robotic researchers, little effort has been made to standardize their APIs, either.

An additional challenge to mobile robot systems is the increased demand on reliability over many typical computer systems. Since robots move around in the real world, hardware or software failures could cause damage to equipment or endanger people near the robot. Robotic hardware is also more prone to failure because of their movement. Robots vibrate, rattle, and may even fall long distances. Hardware failures are to be expected, but the system should degrade gracefully and continue to operate in a predictable manner for as long as possible.

Many mobile robots are large enough to carry a conventional computer onboard for processing. Others are so small that they carry only a wireless communication device and are commanded remotely. While this work has applications for the latter type, our focus is on larger robots, which usually carry a wide array of potentially redundant sensors and actuators. Some devices are particular to mobile robots (e.g. sonars), but many are common computing devices that connect to the internal PC (e.g. video camera). If redundant sensors are available, numerous devices on the same robot may provide the same information, but with differing reliability, sensitivity, or failure modes. Since these devices are usually accessed and commanded through distinct APIs, interchanging them is even more difficult. For example, lasers, sonars, stereo vision cameras, and bump sensors can all provide range information (distance to obstacles), but with varying precision and vastly different APIs. In many situations, if one of these devices were to fail, another could replace it.

2. RELATED WORK

Distributed mobile robot software architectures seem as numerous as the hardware platforms which they support. Although many of their features are common to distributed middleware toolkits outside the realm of mobile robot control, this domain provides enough opportunity for specialization to foster the growth of a fair number of specialized toolkits.

Send correspondence to {nam1,wds}@cse.wustl.edu

The primary features which help distinguish between various frameworks are the mode of interprocess communication (IPC) chosen and the extra, domain–specific software provided with the toolkit. IPC is an important consideration since most frameworks encourage a modular, service–oriented design. Individual device drivers are run in separate processes (or threads), and many data–processing algorithms (e.g. localization or computer vision routines) are run separate from the application as well. Communication must occur between all of these elements. In addition, communication between robots, or between robots and a controlling computer, should make use of primitives provided by the framework. Speed, reliability, robustness, and multiple language bindings are important considerations in choosing a method of IPC.

Added functionality is also an important consideration for these programming toolkits. A general purpose middleware package will provide little more than IPC abstractions, but mobile robot programmers need a toolkit specifically created for mobile robot platforms. Most toolkits provide abstractions and interfaces for various hardware sensors and actuators, interfaces for common algorithms such as localization and navigation, data types such as maps and standard units for measurements, and some notion of coordinate frame transforms. Some toolkits even provide 3D visualizations of the robot and its environment.

The descriptions below provide overviews of some of the most popular mobile robot programming frameworks in use today. Their choice of IPC method and added functionality are described along with some of their goals and design criteria. The features of various IPC methods are also described below.

2.1. Programming Frameworks

Player/Stage: Player[1] is a robot control framework originally created at the University of Southern California for ActivMedia Pioneer 2 robots. It is now developed and used by research labs around the world, and it supports numerous robotic platforms and hardware devices. Although it was originally intended to simply provide interfaces for hardware device drivers, the framework has expanded to provide interfaces for algorithms such as localization. Stage is an accompanying package which provides a 2D simulation environment for Player. A 3D simulation environment called Gazebo was also added to the suite recently.

Since Player was originally designed for a single robot platform, its model for general hardware devices has evolved slowly. The communications protocols for every device must be defined in a single monolithic header file which enumerates the device types and defines message packets for each type of command understood by the devices. Player uses a client/server model, with all communications performed over TCP sockets and bindings for a variety of programming languages.

CARMEN: The Carnegie Mellon Robot Navigation Toolkit[2] is a more recent toolkit which provides modular services with common interfaces. CARMEN also provides pre–built algorithms for person–tracking, map building, localization, and navigation by making use of these common interfaces.

All interprocess communications in CARMEN use a subscription–based model, provided by a separate software package simply named IPC (hereafter referred to as CMU IPC). CMU IPC itself provides a central process which provides a naming service for finding specific services based on their general interface names.

Orocos: Orocos,[3] Open Robot Control Software, is a project primarily developed by Katholieke Universiteit Leuven in Belgium, the Laboratory for Analysis and Architecture of Systems in France, and Kungl Tekniska Högskolan in Sweden, but contributions are received from many European laboratories.

The goals of this project include the development of a real–time kernel for task execution, including execution sequencing, and the integration of a kinematics modelling system. Orocos uses CORBA for its underlying communications primitives.

Miro: Miro[4] is another CORBA–based framework for robot control, based on ACE and TAO.[5] It is primarily developed at the University of Ulm, Germany. The Miro framework provides multi–threading support and a layered client/server architecture that leverages current research in mobile and grid computing on the Internet2 infrastructure.

2.2. Transport Mechanisms

CORBA: Although CORBA (Common Object Request Broker Architecture)[6] tends to be regarded by many as a heavy–handed solution to the problems of IPC on robots, it provides many of the features needed in a service–oriented framework, such as a naming service and transparent distributed operation. The use of an ISO standard Interface Description Language (IDL) permits the creation of objects in any language which provides CORBA bindings.

CORBA depends on the existence of an ORB, or Object Request Broker. The ORB provides the naming service which allows a single point of contact for any communication endpoint seeking to contact another. In service–oriented systems, this usually consists of clients seeking a reference to a particular service. Mobile robot programming frameworks benefit from the added layer of abstraction that class hierarchies naturally produce, so clients may request services with a general interface, rather than requiring a particular implementation.

However, CORBA has been slow to escape its perception as a slow framework encumbered with too many features. In some cases, its general approach to distributed computing has ignored the optimization possible in special cases. For example, many mobile robot perform all of their computation on a single onboard computer. For distributing information to many processes on a single machine, shared memory would be the most efficient solution. Since this is not an option if objects are distributed across multiple machines, it may not be considered by a CORBA–based system.

CMU IPC: CMU IPC[7] was originally designed for internal use at Carnegie Mellon University, but it proved useful in larger projects which were released to the public. It is now available as a stand–alone package. It also provides multiple language bindings and a naming service as a separate process. However, CMU IPC avoids the extra compilation step associated with IDL. Instead, it uses specially formatted strings to describe the content of messages, and runtime method calls to the CMU IPC library allow objects to subscribe to particular message types by means of a callback function. Thus, all messages must travel through the central server before reaching their final destination, but message multicasting is handled automatically. This publish/subscribe model may be more powerful than direct communication between clients and servers, but it requires synchronization in the source code between client function calls and the callbacks that receive messages, since all messages are keyed on string constants.

Ad hoc methods: As in many aspects of programming, there are tradeoffs between the use of separately developed and tested single–purpose packages, and the development of one's own code specifically suited to the task at hand. The middleware packages described above (and many others) have been well–tested in many different computing environments in many different situations. However, ad hoc solutions can be far more efficient even when strict encapsulation is maintained between the IPC layer and the rest of the toolkit. In addition, new features can be integrated in the IPC layer far more efficiently using this approach.

2.3. Standards

Finally, we consider the issue of standardization in robotics. Standard interfaces for device drivers and common algorithms would benefit all robotic researchers and implementors since code could be shared more effectively. Standard file formats would enable sharing of sensor logs and refined information such as occupancy grids and other types of maps. The Robotics Engineering Task Force[8] is a cross–discipline group with members from research and industry, who are interested in these goals. One of the main issues of contention within this group is that of scale. Robots run the gamut from tiny devices with little onboard computation to large machines carrying multiple PCs. Smaller systems are not capable of efficiently converting information from a generic interface to their domain. This issue is not likely to reach a swift resolution. To address the need for standard file formats, though, Radish,[9] the Robotics Data Set Repository, has been established. Its stated purpose is the distribution of data sets (logs and maps) collected by mobile robots. The Radish founders are associated with Player/Stage and CARMEN, so the data sets are available in these formats. It is our hope that the availability of these data will encourage software authors to include support for these formats in other robotics–related packages.

3. SYSTEM DESIGN

3.1. Motivation

With so many frameworks already developed and deployed for controlling mobile robots, the goals of any new toolkit must be clearly defined to properly motivate its creation. An excellent description of the requirements of a robot software architecture was provided by MacDonald et al.[10] We believe that our framework uniquely facilitates the feature set described in that paper, and those enumerated specifically below as necessary for supporting programming on a mobile robot platform. Since our framework provides general facilities for defining, writing, deploying and using distributed services in an unreliable environment, it is equally well–suited as a framework for any autonomic system. However, our purpose is the development of an advanced software infrastructure for robotic systems, so the toolkit provides more than just this framework alone; it includes all the services necessary to operate an iRobot B21r and its attached devices.

The most important feature offered by our system beyond the features of other robot programming toolkits is the integration of recovery oriented computing (ROC)[11] techniques at the lowest level of the framework. Most of the work in control architectures for mobile robots has centered on efficiency, portability, and ease–of–use issues. We are not aware of any work in the mobile robot[12] community that addresses the issues of architectures for fault–tolerant computing. However, a focus on fast and full recovery of the system in the event of failure of multiple components is one of the key aspects of autonomic computing,[13] which our system seeks to support. As described in section 1, component failure is practically guaranteed in mobile robots. While the application programmer needs to keep this in mind, the programming framework should, wherever possible, detect, predict, and recover from failures without intervention from the programmer or the user. The techniques employed by our system to achieve these goals are discussed in section 4, which describes the architecture of the framework, but their motivation should be apparent. Integration of recovery oriented features both enforce their use and speed development. The robot user should not be responsible for diagnosing or fixing hardware and software errors since he is often not qualified, or does not have access to the robot when it is deployed. If the application programmer is burdened with checking every error condition, the code becomes hard to read, and thus difficult to maintain. In addition, this logic must be repeated (often verbatim) in every application. The situation is even worse in code written under time constraints, such as rapid prototyping or bug fixes and feature modifications added during deployment.

3.2. Features

This section describes the desired features of a programming framework supporting autonomic computing. The details of their implementation in our system are described in section 4.

Loosely coupled modules: Structural modularity should be encouraged in the code. Device drivers, common algorithms, and application code should be separated into small, individual processes in order to isolate and diagnose failures. Debugging and unit–testing are simplified since each program only provides a single feature. When a process encounters a failure in either hardware (in the case of device drivers) or software, the process can be restarted or, after multiple failures, entirely removed from the system. Since the process is minimal, the impact of such a change will be minimal. Likewise, when a change is made in one module, only that module needs to be recompiled and restarted. The rest of the system is not affected. This separation also helps conserve resources since the system can determine which modules are needed at any given time, and only run those modules.

Finally, this alternative to monolithic design provides support for programming structures and techniques not otherwise possible. One obvious advantage is that modules can be run on multiple machines. Intensive calculations can be performed on computers not carried onboard the robot, and multiple robots can communicate without additional framework support. In addition, the use of modules suggests an object–oriented development approach which clearly separates and encapsulates the functionality of individual components.

Interprocess communication (IPC): A flexible and robust IPC mechanism is necessary if the entire system is to be flexible and robust. Since our system attempts to handle and correct error conditions without

bothering the programmer, our IPC mechanism must be robust so that it does not become a failure point in the system.

Speed is also an important consideration, particularly in the case of sensors which produce a large amount of information, updated often. The framework should be free to choose the transport mechanism most appropriate to each situation. For example, a service may publish large amounts of information in a shared memory segment for clients on the same machine, but this data may be sent to a different machine through a socket only when requested.

Naming service: Naming is an important aspect in communicating between the modules in a system. A hierarchical naming scheme provides information about the types of devices available in a system, and encourages a form of polymorphism for the distributed objects. For example, a laser range–finder is a particularly common device, found on most mobile robots. However, there are advantages to addressing it as simply a range–finder. Some information is hidden since the particular sensing properties and error modes of lasers cannot be applied to a generic range–finder, but a programmer can write against an application programming interface (API) that is common to lasers, sonars, infrared, and vision–based range–finding devices. Using the loosely coupled modules described above, this programmer can connect to any of these devices without knowing, or needing to know, which one. If one device fails to operate, the failed device can be transparently replaced with another range–finder.

The laser in the example above can be identified by the names RangeFinder, Laser, or even a more specific name that identifies the manufacturer and model number of the laser. This allows users to locate a particular device based on the granularity of the information they require. Using the most general term increases the portability of the code since a target platform need only have a device that provides information of a particular type. If a more specific device name is used, the code will not be as portable, but the programmer can make use of information such as known error modes and advanced configuration commands of particular devices.

Fault detection and recovery: Many programming faults are generic enough to be detected by the framework without additional information from the programmer. The framework should monitor each of its modules and take appropriate action if any process hangs, aborts, or displays abnormal behavior. For example, a memory leak might be inferred in a module that slowly increases its allocated memory over time, so the module should be restarted. If the entire system becomes unstable due to low memory or some other aberrant condition, the system should refuse to refuse to start any new processes. Logging should be a consideration throughout the framework so that a programmer can diagnose any failures whether or not the system corrects them automatically.

The framework should also support more specialized monitoring of particular modules. Modules may be added to the system simply for monitoring and diagnosing error conditions of other modules. These monitoring processes may look for known error modes in devices which can be fixed by simply restarting the device driver. If a device driver is restarted too often, the system may regard that device as broken, and attempt to use a different device instead. This operation should be transparent to user code. If a device fails while a client is attempting to get information from it, the system should replace the device with a new one, and return the requested information.

The framework must also support varying levels of human involvement. Any detected errors and their diagnoses should be logged for a system administrator to read later. Certain errors, such as the failure of a device, may be reported to the user immediately. The system should continue to operate with as much autonomy as possible until human intervention is required.

Rich meta–information: The hierarchical naming structure described above allows similar modules to be addressed by the same name. For this feature to be useful, these modules must provide the same API. Thus, the interface of a module can be determined by the names that can be used to address it. When the programmer defines this interface, semantic information may be attached to each method in the API. That is, rather than simply define the names of the methods and their arguments, the interface should also describe *what* information is represented by that method. For example, a method that returns a

percentage can be described in a more specific way than simply defining a method that returns a float. The meta–information in the interface might specify that the value should be between 0 and 100 and that the best graphical representation for this data is a progress bar. This allows the user to interact with the module at a higher level of abstraction, and permits more intelligent reasoning about each module by automated tools and novice users.

The fulfillment of these features permits the creation of an autonomic system which can:

Self–configure: Once the system knows how to find all of the modules available to it, it can reason about which devices and algorithms to offer when a particular interface is requested. It may profile each module in terms of memory and CPU utilization, and offer different modules based on current memory usage or battery status. Further configuration can result when a module is removed from the system due to errors. The eventual goal is the ability to install the toolkit on a new robot, perhaps provide some minimal configuration and policy information, then allow the robot to manage itself.

Self–diagnose: If additional modules are present for monitoring device drivers, the system should enable these whenever the device itself is in use. The information from these monitors, along with more general system diagnostics, will allow the system to detect all types of errors that may occur.

Self–heal: Once errors are detected, they must be corrected. As previously outlined, failed modules should be replaced by modules with identical interfaces in a manner transparent to user code. In addition, errors may be logged or displayed for the user to see, but human intervention should only be required after more graceful forms of degradation have been attempted and all other options exhausted.

4. ARCHITECTURE

In this chapter, we describe the implementation of our mobile robot toolkit. We will begin by describing the components of the system and defining some terminology, then we will examine these components more carefully. We hope to demonstrate throughout this chapter how the goals of the previous chapter are fulfilled in this framework.

4.1. Components

The four essential parts of the system are Clients, Interfaces, Services, and the Master Control Program (MCP). Each of these components is implemented as a C++ class that provides the necessary functionality. The first three are abstract classes which are extended by the implementor of a device driver or algorithm. Services are equivalent to the modules described in the previous chapter, and they may provide one or more interfaces. Each client can connect to a single type of interface. Note that, although a client–server architecture is employed for communications, we refrain from using the term *server* in order to avoid confusion as to the source (Interface or Service) of the information that each client receives.

4.1.1. Clients

Client is an abstract class which is extended to provide a concrete client for each type of Interface. The Client base class encapsulates the various IPC strategies used in the system and provides all common logic necessary to initiate and sustain a connection between the concrete subclass and an Interface.

Most modes of IPC do not strictly require the encapsulation of IPC method calls in an intuitive API such as a client class. For example, CMU IPC simply relies on the uniqueness of function prototypes, which are registered with a central server. Thus, any application that uses the correct prototype can send data to a server without instantiating a dedicated client object.

However, there are advantages to this encapsulation. Primarily, it crystallizes the concept of a coherent interface by providing a type of contract to the user of the Client: all of the methods in the Client's API must be implemented in any Interface to which the Client can connect, and sequential method calls on a single Client object must be received by a single Interface object. Exceptions to the last part of this contract are permitted

when the Client detects that the Interface can no longer be reached. This situation is discussed below in section 4.2. The logical and structural encapsulation provided by the Client class also aids programmers and automated tools that make use of the Interface. The object oriented structure of application code makes it more legible. During development, we can debug and test services more quickly using interpreted languages, even though all of our clients are written in C++. We use a tool called SWIG[14] to instantiate and make method calls on a Client object from Java, Python, Perl, or Ruby without writing (or compiling) any additional code.

4.1.2. Interfaces

Each subclass of Client connects to a similarly named subclass of Interface. Method invocations on an instance of the Client class operate as remote method invocations on the instance of the Interface class to which the Client is currently connected. An IPC strategy based on the type of method call is used to invoke the method remotely and possibly return a value to the Client. When a Client and Interface are running on the same machine, large blocks of data are accessed by mapping shared memory segments, and commands are sent through named pipes. Commands across multiple machines are sent via TCP sockets, and large blocks of data are sent on demand.

To facilitate the fault tolerant aspects of this toolkit, we encourage interface designers to create interfaces applicable to the most general case possible, and to create additional interfaces to handle commands and data that are specific to particular instances of these devices. We have adhered to this advice in the interfaces that we have written and distribute as part of the toolkit. One example is the interface support for IEEE1394 digital cameras. Rather than creating a single Camera interface, we have chosen to use an Image interface for accessing the pictures published by the camera and a CameraConfig interface to command the camera to change its configuration (e.g. zoom, iris, or focus). Thus, image processing routines can be coded for the Image interface, and they work equally well when connected to a real video camera, a simulated camera, or a test image served by a special–purpose program. No changes are needed in the source code to decide which of these services are used; rather, the user should use the various configuration methods discussed below in section 4.2. The advantage of this approach over other systems is that the special–purpose image serving program does not need to masquerade as a digital camera. It provides the Image interface, which is sufficient for the client. If additional information or controls are needed, the Client may seek to connect to one service or the other. No information is lost by separating the two interfaces of the camera service, but abstraction has been used to provide the right granularity of information in every situation.

In order to distinguish between multiple instances of the same Interface, each implementation provides a qualifier which uniquely names that implementation. This qualifier is appended to the interface name with a colon, as in RangeFinder:SICK_Laser or Image:DC1394_UID_7648af48538b64ed644a. This string is known as the fully qualified interface name.

4.1.3. Services

The service concept is analogous to the module described earlier. A service is a single process, and it may provide one or many Interfaces. In fact, the Interfaces may be dynamic, being created and destroyed as the service runs. To illustrate this flexibility, we will describe a few of the services created as part of the toolkit.

`MCLocalizer` The Monte Carlo localizer is a simple service which provides exactly one Interface: Localizer. This is the only interface necessary for the service to provide its stated functionality.

`SICK_Laser` This service provides two Interfaces: RangeFinder and LaserConfig. The first is a generic interface interchangeable with any other device that provides range information: another laser, sonar, or even a stereo vision algorithm. The second interface is common to all laser range–finders and provides any configuration information typical for lasers. It should be noted that these two Interfaces are implemented in a single service not only because they logically belong together, but also out of necessity. Since the laser range–finder is connected to the robot's onboard PC by a serial cable, only one process will be able to communicate with it. This will be the case for many onboard devices.

`DC1394` This service provides no Interfaces when it is started. Instead, it monitors the 1394 bus for digital cameras and creates and destroys Interfaces as cameras are plugged and unplugged. An Image and CameraConfig interface is provided for each active camera.

4.1.4. Master Control Program

The Master Control Program (MCP) is responsible for starting and stopping Services, as well as responding to Client requests for Interface implementations. Its behavior is controlled by a configuration file and requests received while it is running.

The service concept is important to the MCP since this represents the granularity of control that the MCP has over the system. If a particular Interface is requested by a Client, the service providing that Interface will be run by the MCP, and the MCP can only know which service to run if the service has advertised which Interfaces it provides. This means that the MCP must know about all available services so that it can determine which Interfaces are provided by each service. This information is provided by the configuration file, to which we now turn our attention.

The configuration file contains multiple sections which are known as profiles. Typically, a profile will contain the configuration necessary for a particular robot or test configuration, and will be named accordingly. Many profiles can be defined in the same configuration file, and the MCP can be instructed which to use at startup. After the name of each profile, the command lines for all appropriate services are listed (i.e. full path to the binary, followed by any necessary flags). At startup, the MCP will run each of these services with an additional flag, which will instruct the services to connect to the MCP, list all Interfaces they provide, then shutdown. The configuration file also allows the user to specify the default service to use when a particular Interface is requested by a Client, as well as aliases that permit the Client to refer to an Interface by a different name than the default provided by the Interface itself. This feature is usually used to create a more general name where a specific one is provided by the Interface implementer. For example, a laser device driver, which should implement the RangeFinder interface, may have a default qualifier containing the manufacturer and model number for the laser device. If only one laser range–finder is available on a particular robot, the system administrator may provide a more succinct Laser qualifier for access to that device.

4.2. Client–Interface Interaction

Clients are connected to Interface implementations when the connect method is called on the Client object. This method is preferred to automatic connections in the Client constructor since errors can be reported as return codes rather than as less portable exceptions. The connect() method takes two parameters. The first specifies the qualifier of the preferred Interface to which to connect. This qualifier may, of course, be an alias defined in the MCP's configuration file. The second parameter specifies whether the application should fail or continue if the specified qualifier is not available. Thus, an application may require that its RangeFinder client connect to a Laser if it makes use of operating (or failure) modes particular to lasers, or it may simply *prefer* to connect to a laser since this is the best range–finder device currently in use.

Clients interact with Interfaces in two ways: IPC method calls, and published data. Published data is available through shared memory, and contains data which is too large, or which is updated or polled too frequently to be to be distributed efficiently via sockets or pipes. Clients map the shared memory read–only, so any communication from the Client to the Interface must occur through a socket or pipe mechanism. The current implementation uses FIFOs, but sockets are planned for a future version to permit communication between machines. If a Client looses its connection to an Interfaces while attempting to access it through either of these mechanisms, the connection is automatically restored to the same Interface implementation, if possible, or another implementation, if allowed by the last call to connect(). The details of this recovery are discussed in section 5, but it is important to note that the Client is guaranteed to deliver each method to an Interface, return values from an Interface, or fail immediately.

4.3. Interface Definitions

Finally, we describe the generation of the abstract Interface subclasses and concrete Client subclasses which define the API of services. In order to eliminate the burden of maintaining synchronization between the Interface and Client APIs, all methods and published data are described in validated XML documents. A tool provided as part of the framework translates this document into concrete Client classes for use in applications and abstract Interface classes which can be extended by service implementers. This provides a number of advantages over specifying APIs in source code:

Language neutrality The IPC protocol, with the exception of a standard header, is defined entirely by the XML. Any program which adheres to this protocol is compatible with the framework. While Player/Stage uses constants defined in a header file to distinguish between types of interfaces, our approach uses the name of the interface and a version number. Application and service authors are free to port the thin IPC layer to their favorite language, or use a tool such as SWIG, which provides access to the native C++ methods from Java, Python, Perl, and Ruby. Since our system converts the XML interface description to concrete method calls in the Client class, SWIG users need only call a method with an intuitive name and its required arguments. Player provides similar proxy client classes, but they are synchronized with the IPC layer by hand. Alternately, CARMEN requires calls directly into the underlying CMU IPC layer.

Meta–information XML is infinitely extensible, so extra information can be added to the interface descriptions as the toolkit evolves. The future work section of this document describes the intended addition of tags to specify graphical widgets to best represent certain methods and published data in the interface definitions. Comments in the XML are transferred to the generated source code. As new tags are added for the benefit of newer tools, they can be ignored by the old tools that do not understand them.

Generic APIs Since source code is generated from the XML documents, any alteration to an XML interface description will be reflected in all services offering this interface. This forces interface authors to carefully consider how best to design a generic interface. This is the price of standardization, but it provides the benefits of widespread interoperation. Careful encapsulation allows researchers to share algorithms between different institutions and hardware platforms with the assurance that services and clients speak the same protocol.

5. AUTONOMIC FEATURES

An autonomic system is a system which is capable of governing itself. Our system provides or enables all of the features described earlier as necessary parts of an autonomic system. It can self–configure, self–diagnose, and self–heal.

Given a configuration file enumerating all of the services available in the system, the MCP will determine the interfaces available from each service, and connect clients to them as requested. The list of available interfaces is dynamic since services may be removed due to hardware or software errors, or new services may be registered.

The two information dissemination mechanisms, FIFOs and shared memory, permit two different forms of monitoring, so that Clients may ensure that their Interface is still active. First, if the Interface exits (or crashes), its FIFO will be closed for reading, and attempts to write messages to it will fail immediately. Second, a periodic timestamp, or heartbeat, is published in the header of the shared memory block. If the service hangs, but the process does not terminate, the FIFO will remain open, while the heartbeat will cease to update. The Client will notify the MCP of the error, and the MCP will take appropriate action. The offending process will be terminated and either restarted or unregistered so that it is not run again.

6. CONCLUSIONS

6.1. Implemented Services

The toolkit in its present form contains 22 interface definitions from Blobfinder to Synchrodrive. The package installs 67 binaries, of which 25 are services. Many of these services implement multiple interfaces; one implements as many as five. The rest of the programs are utilities, GUIs, and applications which use one or several clients to control a robot through high–level commands. The toolkit is still in active development.

6.2. Future Work

Our system has been implemented entirely on an iRobot B21r robot, but hardware support for other robotic platforms is one of our primary goals. Support for Pioneer platforms is currently under development, as well as adaptors to provide compatibility between our system and Player. This will also provide us with the use of the Stage and Gazebo simulation environments.

As previously discussed, the choice of XML for defining interfaces allows great extensibility within the toolkit. We would like to provide additional tags along with each method and published data description to specify an appropriate GUI widget for calling that method or displaying its data.

We would like to compare various IPC implementation strategies for speed and simplicity. We have not failed to notice the strong similarities between our MCP and a CORBA ORB. The Washington University Distributed Object Computing group is developing a lightweight, extensible ORB called nORB.[15] We would like to reimplement the MCP as an nORB to determine the impact CORBA will have on the speed and reliability of the system.

Finally, we would like to extend our fault handling mechanisms. Our system can currently detect both generic software faults such as crashes or hangs, and sensor specific faults, but we would like to employ machine learning techniques to predict future failures[16, 17] as well. The system could notify a human operator that a hardware failure may be imminent, or take proactive action to restart a service that begins to behave erratically.

REFERENCES

1. "The Player/Stage Project." `http://playerstage.sourceforge.net/`.
2. "CARMEN: Carnegie Mellon Robot Navigation Toolkit." `http://www-2.cs.cmu.edu/~carmen/`.
3. "Orocos: Open Robot Control Software." `http://www.orocos.org/`.
4. G. K. Kraetzschmar, H. Utz, S. Sablatnög, S. Enderle, and G. Palm, "Miro – Middleware for Cooperative Robotics," in *Robocup 2001: Robot Soccer World Cup V*, A. Birk, S. Coradeschi, and S. Tadokoro, eds., *Lecture Notes in Artificial Intelligence* **2377**, pp. 411–416, Springer-Verlag, (Berlin, Germany), 2002.
5. "ACE and TAO." `http://www.cs.wustl.edu/~schmidt/TAO.html`.
6. "CORBA." `http://www.corba.org/`.
7. "Inter Process Communication (IPC)." `http://www-2.cs.cmu.edu/afs/cs/project/TCA/www/ipc/ipc.html`.
8. "Robotics Engineering Task Force (RETF) Charter." `http://www.robo-etf.org/`.
9. "Radish: The Robotics Data Set Repository." `http://radish.sourceforge.net/`.
10. B. MacDonald, D. Yuen, S. Wong, E. Woo, R. Gronlund, T. Collett, F.-E. Trépanier, and G. Biggs, "Robot Programming Environments," in *ENZCon2003 10th Electronics New Zealand Conference*, (University of Waikato, Hamilton), 1–2 September 2003.
11. D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Mertzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," Computer Science Technical Report UCB//CSD-02-1175, Department of Computer Science, University of California at Berkeley, March 2002.
12. A. Orebäck and H. I. Christensen, "Evaluation of Architectures for Mobile Robotics," *Autonomous Robots* **14**, pp. 33–49, 2003.
13. P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology." `http://www.ibm.com/research/autonomic`, 2001.
14. "Simplified Wrapper and Interface Generator." `http://www.swig.org/`.
15. V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron, "Middleware Specialization for Memory-Constrained Networked Embedded Systems," in *9th IEEE Real-Time and Embedded Technology and Applications Symposium*, (Toronto, Canada), May 2004.
16. B. C. Williams and P. P. Nayak, "A Model-Based Approach to Reactive Self-Configuring Systems," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 971–978, AAAI Press / The MIT Press, 1996.
17. R. Dearden, T. Willeke, F. Hutter, R. Simmons, V. Verma, and S. Thrun, "Real-Time Fault Detection and Situational Awareness for Rovers: Report on the Mars Technology Program Task," in *Proceedings of the IEEE Aerospace Conference*, March 2004.