A Qualitative Comparison of Interprocess Communications Toolkits for Robotics

Jay Gowdy

June, 2000

CMU-RI-TR-00-16

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA

This work was supported in part by Science Applications International Corporation (SAIC)

Abstract

Large robotics projects can no longer be done by a single person writing a monolithic piece of code. Projects are now composed of modules which can be libraries, threads of execution, or stand-alone processes. Different people will almost certainly work on different modules, with the module developers sometimes being geographically distant from each other. Thus, good interprocess communications toolkits to compose these disparate modules into a functioning whole are the fundamental infrastructure of most large robotics projects.

An interprocess communications toolkit abstracts the transport mechanisms, such as shared memory or network protocols, away from the module developer. The goal is for the module developer to use the communications toolkit to write portable code which can be moved from platform to platform and which can integrate with other modules seamlessly, whether those modules are actually implemented in neighboring processes, neighboring machines, or neighboring states.

This paper performs a qualitative comparison of a wide variety of communications toolkits including IPT, RTC, NML, NDDS, MPI, and CORBA. The toolkits are compared based on criteria such as suitability for implementation of typical data flows in robotics, portability, and ease of use.

1 Introduction

Large robotics projects can no longer be done by a single person writing a monolithic piece of code. Projects are now composed of modules which can be libraries, threads of execution, or stand-alone processes. Different people will almost certainly work on different modules, with the module developers sometimes being geographically distant from each other. Thus, good interprocess communications toolkits to compose these disparate modules into a functioning whole are the fundamental infrastructure of most large robotics projects.

An interprocess communications toolkit abstracts the transport mechanisms, such as shared memory, UDP[1], or TCP/IP[2], away from the module developer. The goal is for the module developer to use the communications toolkit to write portable code which can be moved from platform to platform and which can integrate with other modules seamlessly, whether those modules are actually implemented in neighboring processes, neighboring machines, or neighboring states.

This paper explores and compares the suitability of various interprocess communications toolkits as the basic communications structure for robotics projects. The exploration is not through a quantitative analysis with benchmarks, but rather through a qualitative analysis that examines the paradigms and documented implementations of the toolkits and explores how they impact the development and implementation of various kinds of robotic architectures. Section 2 goes into details on the kinds of qualitative criteria that are applied to the various toolkits.

The toolkits to be examined fall into several broad areas. Section 3 examines several message-oriented toolkits in which the basic paradigm is sending, receiving, and processing messages. Section 4 examines several information-oriented toolkits in which the basic paradigm is that modules communicate through shared information data structures. Section 5 examines the preeminent distributed computation toolkit, in which distributed robotic systems may be considered as parallel computational systems, using the traditional approaches from that domain. Section 6 examines the preeminent object-oriented toolkit, in which the basic paradigm is that everything is an object with methods and attributes that can be accessed from anywhere. Finally, Section 7 presents an overview of the toolkits to tie things together.

2 Evaluation Criteria

Once again, the criteria to be applied to the various toolkits are not qualitative benchmarks involving tests of latency and throughput, but more qualitative critiques to deter-

[1] User Datagram Protocol: a network protocol with low latency but no guarantees on delivery.

[2] Transmission Control Protocol/Internet Protocol: a network protocol with delivery guarantees.

mine how applicable a toolkit is to a particular target robot architecture, both in terms of the platforms used and the intended style of dataflow. These analyses result from examination of documentation and code structure rather than from actually implementing experiments.

The goal of these criteria is to form a starting point for choosing a communications infrastructure which reflects the software, hardware, and personnell involved in solving a particular robotics problem.

2.1 Portability

A key question to ask of a communications toolkit is whether or not it will run on the hardware platforms that are available for the task. The best communications toolkit in the world will do no good if it has not been ported to your particular hardware.

Beyond the question of whether a communications toolkit has been ported to a particular platform is a question of how well it makes use of the data transport mechanisms available on the platform. For example, if a toolkit allows processes on the same real-time machine to communicate, will it allow them to use shared memory? Will the toolkit discriminate between UNIX domain sockets and TCP/IP sockets for guaranteed communications under UNIX operating systems? Can the toolkit use the UDP non-guaranteed protocol when appropriate?

2.2 Popularity

The popularity of a toolkit may seem to be a shallow evaluation criteria, but popularity has a large correlation with toolkit usefulness for several reasons. A more popular toolkit has a larger user pool, and the larger the user pool, the more well tested and honed the toolkit will be, and the more likely that the toolkit provides significant benefits to the system developer. In addition, a toolkit's user pool serves as a resource for help and advice with problems: The bigger the pool, the more likely that someone in the pool can help with any problems that may arise in system development.

The ultimate in popularity for a communications toolkit is to be part of a standard. To be a standard means that there is some official authority maintaining the integrity of the toolkit and providing a means for structured, ongoing improvement. Besides improved stability, adopting a standard communications infrastructure raises the possibility of leveraging off of third party software which shares the standard. Such leveraging can be highly effective in producing impressive systems quickly, since the system developers can use existing software to provide system functionality rather than being forced to develop it all from scratch.

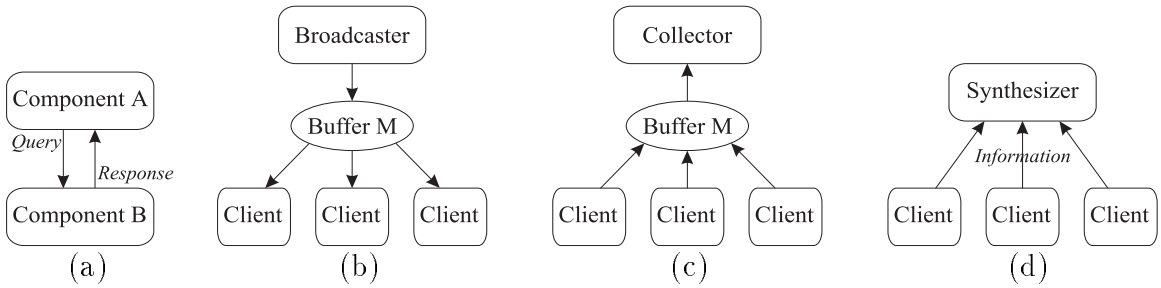

Figure 1: Typical data flow patterns in robot systems: a) Query/Response, b) Broadcast, c) Collection d) Information Synthesis

2.3 Reconfigurability

Different toolkits will support different levels of system reconfigurability. Some toolkits will allow components to come and go at run-time, independent of where the components are physically located. Other toolkits may expect a more stable system, where the components and component topology are specified ahead of time.

2.4 Suitability

Not every communications toolkit is suitable for every robot architecture. The suitability of a toolkit is not directly how efficient it is at pushing data around, but more how naturally the communications abstractions that the toolkit presents to the user will map onto the expected dataflow of the robot architecture. The suitability of a toolkit for a robot architecture may translate into efficiency, but, more importantly, it directly correlates with the required developer effort to use the toolkit to implement the given robot architecture.

There are four common types of data flows in robot systems that will be used in this paper as an aid to evaluating various toolkits. Different robot architectures will use varying mixes of these four types of data flows, with emphases on different ones for different architectures. The four types (also illustrated in Figure 1) are,

- *Query/Response.* A software component A sends a message to software component B and expects a response. This expected response could be a return message, a change to B's internal state, a physical action, or simply passing a message along to yet another module. An example could be a command from a client to a vehicle controller to activate the left turn signal. Messages going from A to B must be guaranteed to arrive, or the system will not function properly.

- *Broadcast.* A broadcaster constantly updates a memory buffer M. Clients can attach to the memory buffer and view its contents. A common example of this kind of data

flow is the propagating of vehicle state from the controller to modules that require it. The state is expected to be continually updated, so if a client occasionally misses an update, the system should continue to function properly.

- *Collection.* A collector maintains a memory buffer M. Clients can attach to the buffer and change its contents to effect some operation of the collector. A common example of this kind of data-flow is a controller which maintains its goal point in its configuration space in a collection buffer, with one or more clients continually updating the goal to move the robot. Since the goal is continually updated, if the server misses a few, the system should continue to function properly.

- *Information synthesis.* An information synthesizer collects information from any number of clients. Only the most recent information from each client is important, but each clients information stream must be kept separate from the other clients information streams, since the server does the synthesis itself. An example would be a steering arbiter, which gets "votes" from each client for where to turn and combines those votes into a single output. We expect a constant stream of information coming from the clients, so if the server misses some, the system should continue to function properly.

2.5 Ease of use

How easy a toolkit is to use depends in large part on how familiar the developers are with it or with similar toolkits: if members of the development team are familiar with the toolkit's paradigms, then the tool kit will probably be fairly easy to use for that team. Even so, there are some estimates that can be made for how hard a particular toolkit is to learn, i.e., on average what is the learning curve for a naive user picking the toolkit up and attempting to perform system integration with it.

3 Messaging Paradigm

In a toolkit working under the messaging paradigm, the basic unit of information is, logically, a message. A message is an atomic unit with a message type and a string of bits containing the data. The message is sent directly from one module to another using a guaranteed method of delivery.

While the details of the toolkits may vary, the messaging paradigm is most useful in event driven, asynchronous system architectures. In these architectures modules are driven by events and incoming messages. The modules wait for an event (which is often an incoming message) and respond immediately to that event. Messaging toolkits

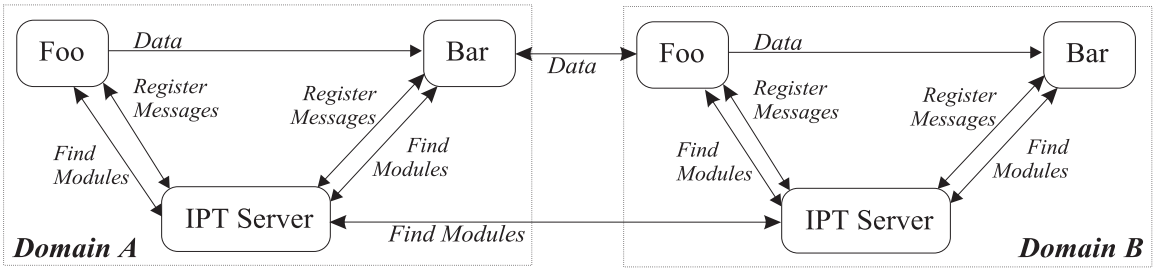

Figure 2: IPT architecture

have a variety of support mechanisms for integrating modules in such an event-based architecture.

3.1 IPT – InterProcess Toolkit

The InterProcess Toolkit (IPT) is one such messaging based communications toolkit designed for robotics research [1]. It was developed and written in C++ (with hooks for C) for use in the Unmanned Ground Vehicle program [2]. The structure of the systems in the UGV program were that they ran almost entirely on networked workstations under UNIX, with very few processes running on real-time systems.

IPT was designed for simple, point-to-point message based communications. The IPT server exists to provide two basic functions: First, to connect modules together using a simple routing table so that modules can be referred to by names rather than machines and ports, and second, to provide a central space to convert message names into more compact but unique message numbers. Once two modules connect together and register messages, they can ignore the server entirely, sending their messages back and forth to each other. There is provision in IPT for multiple IPT servers, with each IPT server servicing a "domain." The inter-server communications was optimized for low bandwidth links, since the design was to have an IPT server in each robotic vehicle with low-bandwidth wireless links between the vehicles.

IPT has been ported to many non-real-time, UNIX based, operating systems, such as SGI's IRIX, various flavors of SunOs, Solaris, and Linux. In addition, there has been a relatively untested porting to some real time systems such as VxWorks and LynxOS. IPT's internal architecture is fundamentally object-oriented, so, for example, it has different connection classes for different transport mechanisms. Thus within the IPT framework it is straightforward to use TCP/IP sockets for inter-machine connections and UNIX domain sockets for intra-machine connections. There does exist a connection class implemented under VxWorks which uses shared memory to communication between processes that exist on machines sharing a VME backplane, but this communications mode is relatively untested.

IPT has a rather small user pool which is essentially limited to the researchers involved in UGV program. It has not become any kind of industry standard, and thus an IPT based system will be difficult to integrate with third-party software.

IPT is designed to be fairly flexible in managing its connections. Each client provides a name for itself and an ordered list of ways in which it can be reached, or "routes", and when another client requests a connection by name, the server attempts to connect the two clients together with the dynamically determined best connection method. For example, the server would recognize from two modules routing lists that when they are running on the same UNIX machine, they can be connected together with UNIX domain sockets. When the same two modules are running on different UNIX machines, the routing is through TCP/IP sockets. IPT provides the clients with the ability to detect when modules connect and disconnect, so there can be some measure of dynamic structure to a system, but typical use is to have a fairly stable structure for the duration of a single test.

IPT has been finely honed for the Query/Response model of data flow in robotic systems. It provides a variety of methods for modules to handle incoming messages and respond to them. This kind of data flow is exactly what the message passing paradigm is designed to work with.

The broadcast model presents a problem. In IPT, as in other messaging toolkits, there is no abstraction for a shared memory buffer: The information dissemination must be done via messages to each client. In the development of IPT, it was recognized that this type of information dissemination is a common mode of data flow in robotic systems, so it has a set of publisher/subscriber primitives which implement the equivalent functionality as a shared memory buffer readable by clients but writable by the server. In this approach, the server declares it is "publishing" a message type and clients connect to the server and "subscribe" to that message type. When the server publishes a message, it is automatically sent to each client.

The messaging paradigm is not the most suitable for this type of data flow. When clients and servers share an address space, the message paradigm forces "messages" to be sent to each client, even though the same data could be propagated by updating a single physical shared memory buffer protected by semaphores. When clients and servers do not share an address space, the message paradigm forces the use of guaranteed delivery mechanisms such as TCP/IP or UNIX sockets when the circumstances allow more efficient broadcasting mechanisms such as the UDP protocol.

Every change to one of the "buffers" represents a message in IPT to all the consumers of that information. In a typical scenario, incoming messages get put in a queue for handling by the consumer. If the consumer spends significant portions of its time in computational processing, then there can be a significant backlog of messages to deal with by the time the consumer is ready to process new messages. In the worst case design, the user program handles and processes each of these queued messages, thus

falling further and further behind the incoming data stream. A better design is for the user to go through the message queue and ensure that they are processing the most recent message, discarding the others.

IPT provides a mechanism to alleviate the problem of backlogged messages by minimizing the message queuing and data processing that the consumer process would otherwise incur. In IPT's object-oriented scheme, each message type can be given a "destination" object which determines what to do with the raw incoming data. The default destination object makes a copy of the message and puts it on an internal message queue. For broadcast message types the normal formatting and dispatching of message data can be circumvented so that the incoming data is simply put into a buffer. As new information comes in, the buffer is overwritten.

Similar problems arise for IPT and other messaging paradigm toolkits for the "collection" mode of data flow, in which, in an abstract sense, there is a shared memory buffer which is writable by clients and readable by a server. Once again, IPT must implement this dataflow pattern with guaranteed messaging, but it can provide the same kinds of optimizations for reducing message backlogging that are provided for the broadcast case.

There are some similar inadequacies of the messaging paradigm used in IPT when the toolkit is used for information synthesis, in which any number of clients are sending streams of a particular message type to the server, and the server synthesizes the latest value of each source to produce a single output. The abstraction of the data sources as streams is closer to the underlying message paradigm used by IPT, but IPT still forces the use of guaranteed delivery transport mechanisms when the less reliable, but more efficient, transport mechanisms may be more appropriate. Once again, IPT's "destination object" abstraction can be used to optimize the handling of the data, funnelling the data from each client to the appropriate buffer which represents the latest information from that client while minimizing message queuing and processing..

IPT is a fairly simple communications toolkit. It was produced by a single developer given the pressures of the set of particular system integration tasks within the UGV program), so it has remained as simple as possible, while still satisfying the needs of that particular domain. It is not very hard to do basic communications in IPT, but it has some subtleties, such as the "destination object" abstraction, which can be hard to master.

3.2 RTC – Real-Time Communications

The Real-Time Communications (RTC) toolkit is very similar to IPT [3, 4]: It is a message based toolkit in which modules registers its name with a server and which queries the server for the locations of other modules by name, so there is the same kind of run-time flexibility and reconfigurability as IPT. Messages are sent point-to-point directly between modules once the server connects modules together.

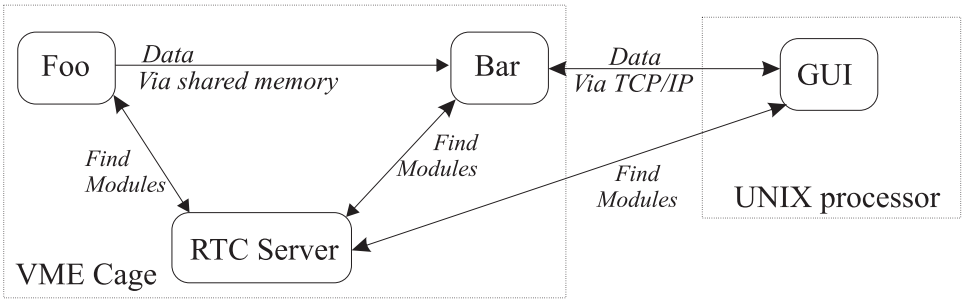

Figure 3: RTC architecture

The designer of RTC had previously used IPT, and some of the differences between the two arise out of a desire to reduce toolkit complexity in areas the RTC author found to be irrelevant for his tasks. For example, in RTC the server is only for registering module names: each message type has a name, but only for a module's convenience, and it is up to the system developer to assign message numbers in a consistent manner across modules. In addition there can only be one server in a system: There is nothing like the concept of "domains" found in IPT, since the systems the RTC developer was working with had no need for them.

Other differences between RTC and IPT arise out of subtle differences between the typical system architectures in which they are used. In IPT, the typical system architectures involve processes distributed over UNIX workstations, with only a few modules running on real-time operating systems. With RTC, the typical system architecture has most processes distributed over real-time processors sharing a VME backplane, with only a few "auxiliary" interfaces and logging systems implemented on external workstations. Thus RTC has a two-tiered system for messages: High priority messages which have been optimized for sending between real-time processes via shared memory and regular priority messages which will be transported via network protocols. An RTC user explicitly declares a message as high priority, or in the parlance of RTC as a "lightning link," with the understanding that if the message goes between processes which can access shared memory, it will use shared memory in order to maximize the communications resources available.

RTC does not concentrate on maximizing the efficiency of socket based communications, as evidenced by the sole reliance on TCP/IP socket protocols rather than allowing UNIX socket protocols to connect processes which run on the same UNIX workstation (UNIX domain sockets are 2 to 3 times more efficient than TCP/IP sockets in this situation). Intra-UNIX machine communications is common in typical IPT systems, so the IPT developer made sure that the communications resources were maximized for it while paying scant attention to communication between processes running in a real-time environment. The different "typical" systems for which the two toolkits were developed result in different implementations with different strengths and weaknesses.

RTC was developed for work done in the Field Robotics Center (FRC) and the Robotics Engineering Consortium (REC) at Carnegie Mellon's Robotics Institute. It is written in C and is finely tuned for VxWorks, as well as integrating with roughly the same mix of UNIX-based systems as IPT.

RTC has a slightly larger current user pool than IPT, with several current FRC and REC projects utilizing it. It is in no way an industry standard, though, and has few users outside of Carnegie Mellon. It has been ported to Java, allowing a measure of interoperability with web browsers.

As with IPT, RTC is well designed for the Query/Response data-flow. RTC has a slightly less sophisticated data handling mechanism than IPT, but it is sufficient for most cases.

RTC has the same fundamental difficulties implementing the broadcast and collection dataflow patterns as IPT, as it is still fundamentally a message passing system. RTC also attempts to assist and optimize the data handling of these flow patterns. As with IPT, RTC provides a publisher/subscription system that handles the distribution of data. Rather than implementing a generic "destination object" as IPT does, RTC provides the ability to declare fixed length queues for incoming messages. By declaring a queue length of 1, the user is guaranteed to have the most recent message with a minimum of data copying and queueing.

RTC's simplified message handling system does present some disadvantages for the information synthesis case. RTC's message handling system will efficiently provide the most recent message of a given type, but it will not provide the most recent message from each client of a given type. Thus, RTC forces the user to queue and process all incoming messages and pick out the messages of the most recent type from each module, thus increasing the likelihood of problems due to message backlogging.

RTC is definitely simpler to master and understand than IPT, but that simplicity comes with some flexibility costs. If a target system architecture, both hardware and data flow, is very similar to the architecture that RTC was designed for, then it is a better choice than IPT. If more general data handling is required, or the architecture requires more intensive data transfers between processes on the same UNIX machine, then IPT is a better choice.

4 Information Based Paradigm

In a message-based paradigm, the abstraction is that there are direct connections between modules that serve as conduits for messages. In information based paradigm, there is no real "connection" abstraction. The connection between modules is in the abstract concept of an information type. This information type may be represented as a buffer which is shared between many modules, (see Section 4.1), or it may be represented as an

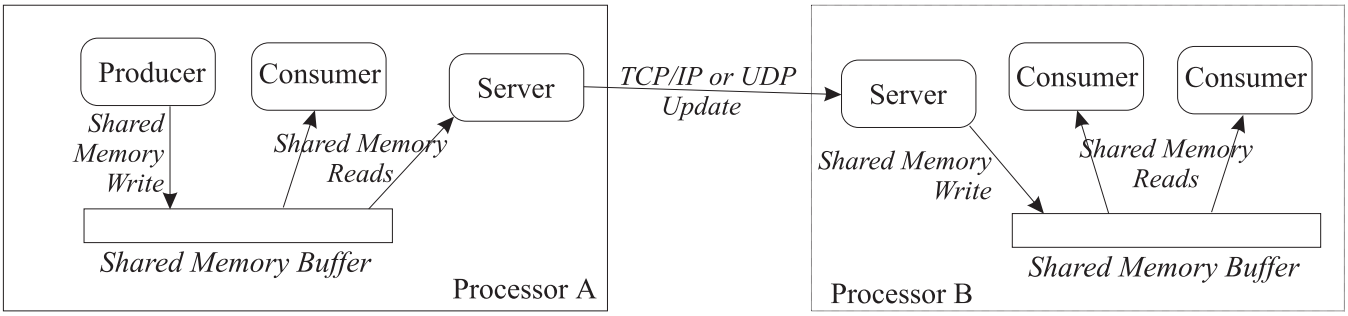

Figure 4: NML architecture

information name which some modules produce and other modules consume (see Section 4.2). In an information based-system, producers of data tend not to care who consumes that data, and consumers of data tend not to care who produces it. The producers and consumers come together anonymously around the information, rather than the system designer thinking about information explicitly being sent from producers to consumers.

4.1 NML – Neutral Messaging Language

The Neutral Messaging Language (NML, also known as the Neutral Manufacturing Language) is an extreme example of an information-based communications toolkit [5]. The basic unit of communications in NML is the shared memory buffer. Producers of information put their data into output buffers while consumers attach to those buffers to read the data. The buffers can be implemented using actual shared memory when possible, but when processes do not share an address space the shared memory buffers are simulated via network protocols. Figure 4 shows an example of data being propagated both by shared memory and through network protocols such as TCP/IP or UDP.

NML has been designed by the National Institute of Standards and Technology (NIST) almost exclusively for synchronous, time-based robotics systems written in C++. Whereas a module in an asynchronous system waits for an event or message to react to, a module in a synchronous system works on a fixed clock. When the clock wakes it up, it examines the input information structures it has access to, performs some computation, places the result in output information structures, and goes back to sleep. Not surprisingly, NASREM, the standard reference architecture for robotics proposed by NIST [6], consists of a hierarchy of rigidly synchronous layers, and thus is perfectly suited to use NML as its communications toolkit.

An NML system is designed to be reconfigurable, but not at run-time. Rather than having a server which at run-time computes how to connect the various components together, NML has the user provide a configuration file which specifies exactly what processes will run in the system, which of those processes "own" data buffers, and how

other processes connect to those data buffers. The configuration file allows for an amazing level of optimization of the system communications, but at the expense of fixing the system configuration and requiring the user to generate a fixed, detailed map of the expected communications in the system.

NML is able to use a large number of transport mechanisms for modules accessing the data buffers, with several kinds of shared memory available, remote procedure calls, TCP/IP sockets, UNIX sockets, and even UDP sockets when needed. In addition, NML has been ported and tested on a wide variety of real-time and non-real-time operating systems. Thus NML gives the freedom to use almost any combination of hardware while allowing the user to precisely optimize the data flow of their particular system.

NML has a fairly large user pool, and can be considered an industrial standard, since it has the support of NIST. In fact, NIST has made a concerted effort to make NML systems open to third-party software, such as web browsers running Java, thus easing integration of NML systems into a larger context.

NML uses the most efficient paradigm possible for the broadcast and collection data flow patterns, since its real implementation precisely matches the communications abstraction shown in Figure 1b and Figure 1c. Where the transport mechanism can be shared memory, NML provides the shared memory mechanisms. Where there must be network protocols involved, NML provides a wide variety, including UDP for maximizing efficient transport of data which doesn't absolutely have to be guaranteed to arrive. NML completely avoids the message backlogging problem by using an external server to process incoming messages and write them into shared memory accessable by consumers and producers.

NML's paradigm starts to break down for information synthesis. Remember, in information synthesis the source of the data is as important as the data, since the algorithm is to combine inputs from various data sources producing the same type of data. NML will force us to have a separate data buffer for each data source. One implementation path is to have the synthesizing module own a fixed number of buffers for input, and information sources have to sign up to use a particular one. NML will provide transport mechanisms for the data to flow between information sources and the synthesizer that are potentially far more efficient than those used in the corresponding message based toolkit, but this efficiency comes at a cost of inflexibility and difficulty of rapidly switching between information sources for testing and debugging purposes. In addition, NML forces the arbiter to be synchronous: it is very difficult to built an synthesizer on top of NML which calculates a new output as soon as it receives input from a data source: the natural structure of an NML based synthesizer is to periodically poll its input buffers and produce an output.

NML's paradigm almost completely fails for use in implementing the query/response data flow. NML provides some mechanisms to make it possible, such as allowing shared buffers to contain queues of incoming changes, and in addition some extensions to NML

have been implemented to ease the pain of implementing query/response data flow [7], but these efforts do not eliminate the basic mismatch between NML's approach to data and the needs of a system that uses queries extensively. NML has no ability to react to incoming data, so clients have to poll for incoming data. NML's data mechanisms make it difficult to trace back where data came from, so users have to build protocols on top of NML to track how to respond to it. NML, by its very nature, is simply not suitable as a toolkit for an asynchronous robot architecture.

NML is fairly simple to use since it has a very simple data flow abstraction. Unfortunately, the simplicity of use comes at the cost to the user of having to write a configuration file specifying the exact relationship between every process and every shared data buffer in the system. This approach allows great precision in optimizing data flow in the system, but that precision results from a tedious, error-prone chore that the system integrator is forced to do by hand.

4.2 NDDS - Network Data Distribution System

The Network Data Distribution System (NDDS) is an information based communications toolkit based on a more abstract model than NML [8, 9]. In NDDS there are information producers, and information consumers. Producers output information of a given type anonymously, while consumers subscribe to information of a given type anonymously. Producers don't know where the data is going and subscribers don't know where their data is coming from: all of that is abstracted away by the basic communications model.

As the name applies, NDDS assumes that its various modules are connected by a network, and all communications is done via UDP. Each processor in an NDDS system runs an NDDS agent which acts as a broker for information types. The NDDS agents are told what other machines are in their "peer group", and information subscriptions and publishings are transparently transported across the processor barriers within the NDDS peer group. When a module declares itself as a consumer of a type of information, the NDDS agents put it in direct contact, via UDP, with the publisher or publishers of that type of information.

The structure of NDDS means that it is very flexible at run-time, allowing modules, or even machines to connect and disconnect at will during the operation of the system. Thus we see a tradeoff between NDDS and NML, where NDDS has exactly one transport mechanism, but creates extremely flexible systems at run-time, while NML has a myriad of transport mechanisms but is rather rigid at run-time.

NDDS was designed as a laboratory tool rather than a robotic integration system. Its design is perfect for a system of networked instruments, each of which is producing data that remote systems are analyzing.

NDDS is a commercial product from RealTime Innovations, so unlike the other tools

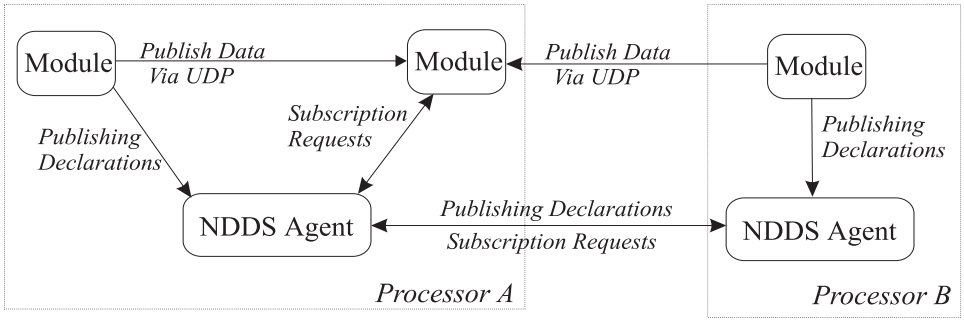

Figure 5: NDDS architecture

that have been reviewed here, source code is not available and there is a monetary cost for non-academic usage. Since it is commercial, it has been ported to a wide variety of operating systems and is well supported and integrated with other useful products from the same company.

Although NDDS' basic transport mechanism, UDP, does not guarantee message arrival, NDDS does implement a protocol on top of UDP to ensure delivery of messages when the user desires that feature. This functionality, along with some rudimentary handling of subscribed messages provide the necessary components to implement much of the query/response dataflow. Since subscribed messages are anonymous, i.e., the publisher is unknown, the user will have to encode the message source in the message in order to be able to reply to the appropriate module if necessary.

The broadcast and collection data flows are where NDDS can shine in comparison to message based paradigms: UDP allows a much more efficient propagation of data from publishers to subscribers than TCP/IP in order to broadcast and collect periodic data. NDDS does not provide any mechanisms to communicate via shared memory, so the data transfer between modules that can share address spaces is not optimized, but NDDS was not designed with this in mind: It was designed to deliver data over a network, not over a VME backplane.

The information synthesis dataflow presents problems for the publisher/subscription approach that NDDS uses. In the publisher/subscription paradigm, the publishers are anonymous, but in an information synthesis, the source of the information is critical. Thus, in an NDDS system the messages used in information synthesis must encode the source of the information, and the "subscriber" of the information to be synthesized must carefully examine every message which comes in in order to parse the information into the right "box" corresponding to the correct source, thus introducing the message backlogging problem seen in the message based toolkits.

NDDS only has one underlying transport mechanism and is extremely pure about its publisher/subscriber paradigm, and thus it is a fairly simply communications toolkit to use. Unfortunately, its purity of paradigm means that it is sometimes difficult to get

NDDS to do exactly what is required, and it misses some opportunities for data flow optimization.

5 Parallel Computation Paradigm

Some of the earliest usages of interprocess communications were for support of parallel computing. In a parallel computing problem there is one algorithm which is distributed across many computers. Many different toolkits arose to support parallel computing in heterogeneous domains, so that, for example, a distributed algorithm could be developed with components communicating using TCP/IP between several workstations and seamlessly ported to a multi-processor super computer.

Many toolkits to support distributed programming have been developed, but MPI, the Message Passing Interface, has emerged as a consensus choice as a standard for interprocess communications in this domain [10, 11]. As a de facto standard with numerous implementations, MPI has a huge user pool and is stable and efficient across a wide range of platforms and transport mechanisms.

MPI and its brethren have been honed for implementing distributed components of a single algorithm, but it is unsuitable for implementing robotic systems, in which the goal is to integrate different components, each of which represents its own algorithm. MPI is not unusable for robotics, just unsuitable in its raw form.

MPI is based on messages, but the messages in MPI are less structured than the messages of IPT and RTC. MPI messages contain only data, and there is no inherent concept of a message name or classification. The basic primitives for message passing are sending and receiving of data structures from process to process, with more sophisticated primitives for gathering and scattering data streams to and from distributed processes. To implement the basic data flow patterns used in robotics would take an entire structure of protocols riding on top of MPI to add the ability to send, receive, dispatch, and handle named messages. In fact, one intriguing idea is to use MPI as the transport layer for a message based system such as IPT, thus instantly increasing that message based system's portability and efficient usage of communications resources.

6 Object Oriented Paradigm

In an object-oriented communications toolkit, the basic unit of information is not a connection, or message, but it is an object. An object is a software entity which encapsulates a set of methods (routines which can be invoked) and attributes (values that can be read and set). Their are numerous other official properties of objects having to do with abstraction and inheritance, but encapsulation is the one we will focus on here [12]. A user

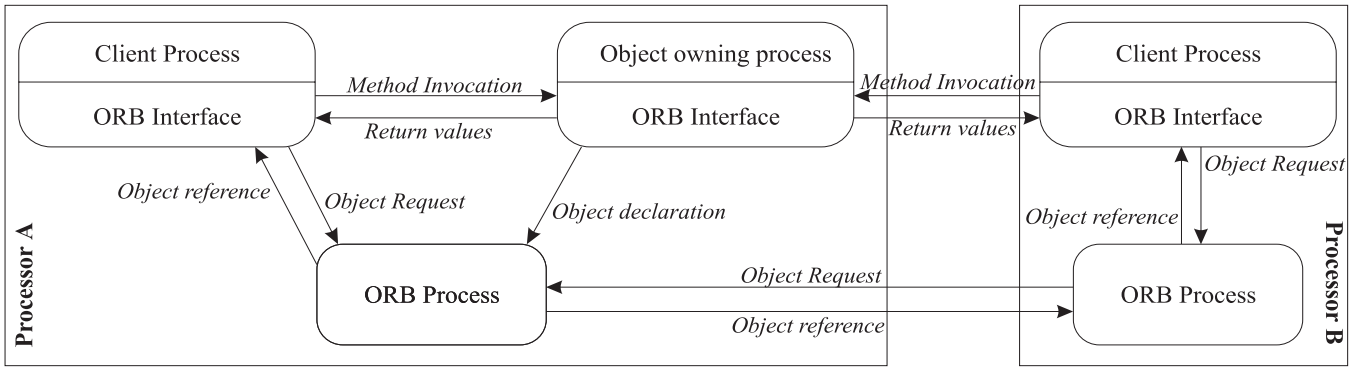

Figure 6: CORBA architecture

will obtain a reference to a remote object and execute methods of that remote object. These methods may pass parameter values to the remote object and may return values from the remote object. As far as the user is concerned, the object being manipulated is in its local space with all of the interprocess communications being hidden behind the abstracted object stub that the user is actually manipulating.

The Common Object Resource Broker Architecture (CORBA) is the de facto object oriented communication standard for non-Microsoft products, and is implemented for Microsoft operating systems as well [13]. CORBA is simply a standard monitored by the Object Management Group (OMG) [14], and there are many implementations, some commercial and some freely available. At the heart of CORBA is the Object Request Broker (ORB). ORBs connect clients to objects, i.e., the broker the relationship between objects and their users. Note that in real systems there is not simply a monolithic ORB process through which all transmissions happen: As Figure 6 shows, the ORB functionality is distributed so that only transactions involved in locating objects go through a separate ORB process, while communications between the users and the objects happen using a portion of the ORB embedded in the user process itself which performs point to point communications using guaranteed delivery protocols.

Since CORBA is a wide-spread, relatively mature standard, it has a huge user pool with numerous resources available for learning and advice. In addition, there is a large amount of third party software which integrates with CORBA. For example, every modern web-browser has a CORBA ORB with Java hooks built-in. CORBA comes from the generic world of software engineering, and in the past has not been appropriate for real-time computing tasks. Fortunately, in the last few years there has been a push to develop a real-time CORBA standard [15]. In fact much of this push has been the result of a freely available ORB modified for real time tasks developed at Washington University (St. Louis) called TAO (The ACE Orb, where ACE is a set of object-oriented wrapper for their transport layers) [16, 17]. TAO has been ported to almost every real-time operating system in common use as well as to most non-real-time platforms and purports to make efficient use of shared-memory protocols for processes that share an address space as well

as TCP/IP and UNIX sockets and remote procedure calls for communications between networked machines. TAO has a long history, and has recently become commercially supported (although it remains open source and freely available) [18].

CORBA is designed for systems that are extremely dynamic at run-time. Objects can be found anywhere and have their ownership moved from machine to machine. The particular CORBA implementation is responsible for picking the optimal transport mechanism for a given object at run-time, although the user can give hints in their code for how they want to access the objects.

CORBA is extremely well adapted for the query/response data flow. When an object is contacted, invoking its methods is equivalent to sending that object's process a message. The invoker can then wait for results in the form of a return value or expected side effects of the method invocation. A CORBA system does not require the elaborate web of callbacks seen in the message based toolkits on the "server" side, since the action to be taken when a method is invoked is defined in the server's implementation of the object class.

Simply relying on the method invocation paradigm could cause some serious message backlogging problems for CORBA in implementing the other common robotic system dataflows, but, unlike most of the other packages reviewed here, CORBA goes beyond its fundamental paradigm whenever it is necessary. With CORBA comes CORBAservices, which extend the abilities of CORBA. These services include such items as Trading Services for locating objects by description, Security Services for communicating securely, Name Services for locating objects by globally distinct names, etc. One CORBA service which is particularly useful in robotics is the Event Service.

The CORBA Event Service allows developers to use the same information-oriented publisher/subscriber model for data transmission that is used in NDDS (see Section 4.2). Users can publish and subscribe to events in "event channels," which route events from publishers to subscribers. The TAO real-time implementation extends the standard Event Service specification to allow the event channel to filter events, so that subscribers can specify they only want events of a range of types or from a certain source. The filters can even be correlative, i.e., they can watch for conjunctions of events and only pass those events to subscribers when the appropriate conjunction occurs. In addition there has been extensive work in TAO to make the dispatching of events as efficient as possible, minimizing data copying and allowing the use of UDP for the propagation of events.

The broadcast and collection data flows are easily implemented with TAO real-time event services. In the broadcast data flow, there is one publisher and many subscribers. In the collection data flow there is one subscriber and potentially many publishers. In these data flows, the event channel is a mediator between the publishers and subscribers. This mediation provides flexibility, but it also potentially increases the overhead of getting data from publishers to subscribers. This may be especially critical for applications in

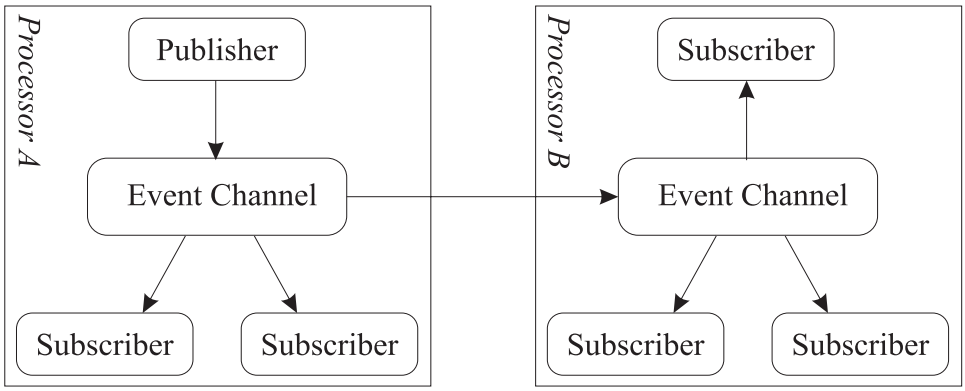

Figure 7: CORBA real-time Events Service implementation

which processes share an address space, and going through a mediator will represent a significant reduction in efficiency compared to simply copying into a shared memory buffer protected by semaphores. On the other hand, mediation has the potential for increasing efficiency for network communications. First, there is the potential for using efficient protocols such as UDP, but more importantly, as Figure 7 shows, the event channels can be transparently federated across machines, so that rather than a publisher having to send many network messages to remote subscribers collocated on the same machine, the event channel can be split up across the machines so that only one message goes between the two event channels, and the event gets propagated using efficient intra-machine transport mechanisms to the subscribers on the remote machine.

The information synthesis case represents the same challenge for CORBA with the event channel paradigm as for NDDS The paradigm assumes anonymous publishers and subscribers, so information about the source of the data must be encoded in the messages, and the receiving synthesizer must go through all the events received in order to obtain the most recent set. Synchronous synthesizers can let the event channel do much of this work by acting as "pull" subscribers: every time they "wake up" they can pole the event channel using a filter for the latest event from every source. Asynchronous synthesizers must take the data as they come and do any filtering themselves.

CORBA and TAO promise to solve almost all of a system developers current and future problems. Unfortunately, CORBA is so comprehensive that it is a significant challenge discover, let alone master, the correct subset of capabilities and optimizations that are needed for any given integration task. There are numerous portions of CORBA and TAO that any given robot system developer will never use, but all of these various "irrelevant" aspects of CORBA and TAO must be kept in mind as the scope and extent of tasks change.

	Real Port.	Non-real Port.	Real Trans.	Non-real Trans.	Popularity
IPT	4	7	2	6	1
RTC	4	7	3	5	2
NML	10	9	10	10	7
NDDS	5	8	1	7	6
MPI	5	10	10	5	9
CORBA	10	10	9	10	10

	Reconfig.	Query/Resp.	Broadcast	Collection	Synthesis	Ease of Use
IPT	6	10	4	4	7	8
RTC	6	10	3	3	6	9
NML	3	1	10	10	8	6
NDDS	8	6	8	8	5	9
MPI	4	2	2	2	2	5
CORBA	10	10	7	7	7	2

Figure 8: Comparison of various toolkits

7 Overview

Figure 8 presents a summary of the qualitative analysis of the various communications toolkits. The ratings range from 1 (lowest) to 10 (highest) and are fairly subjective from the point of view of the author, but reflect the discussion carried on above regarding each toolkit. The keys are as follows,

- *Real Port.*: Level of portability to real-time systems.
- *Non-real Port.*: Level of portability to non-real-time systems.
- *Real Trans.*: How well transport mechanisms on real-time systems are utilized.
- *Non-real Trans.*: How well transport mechanisms on non-real-time systems are utilized.
- *Popularity*: Level of popularity.
- *Reconfig.*: How dynamically reconfigurable systems using the toolkit are.
- *Query/Resp.*: Suitability for the Query/Response data flow pattern.
- *Broadcast*: Suitability for the Broadcasting data flow pattern.
- *Collection*: Suitability for the Collection data flow pattern.
- *Synthesis*: Suitability for the Information Synthesis data flow pattern.
- *Ease of Use*: How easy the toolkit is to use.

Some communications toolkits, while very popular and appropriate in one domain of computer science, are just not suitable for the typical data flow patterns exhibited by robotics systems. For example, in order to use MPI as a robotics communications toolkit,

either the robotics system must be reformulated as a single distributed algorithm, or the system integrator must add enough structure and protocol on top of MPI to use it to express the normal flow of data in a robotics system.

No one existing toolkit is best for all robot architectures. For example, if the robot architecture that is being designed is heavily synchronous, with modules sampling their inputs and producing outputs at constant clock rates, then there is little doubt that the best choice for a robotics toolkit is NML. As more asynchronous, event driven influences affect the architecture, NML becomes less effective at expressing the necessary data flows.

Even when the demands of an architecture specify a certain class of toolkit, logistical issues may lead to the choice of one toolkit or another. Toolkits such as IPT or RTC are the work of a single developer, and can easily be completely understood by a single developer. If IPT or RTC are used to integrate a robot system, then every aspect of that systems communication can be completely understood by a single integrator. That level of understanding is a great benefit to system integration, but it comes at a cost: The packages are designed for specific types of systems perform specific types of tasks. If the target system deviates from these types of systems and tasks then the communications toolkit has to be rewritten or replaced to best implement the new system. To contrast this situation we can look at toolkits such as CORBA which are the results of hundreds of developers working over many years. It is difficult to expect any single person to understand every aspect of the communication occurring in a CORBA system, even with a CORBA implementation that provides the source code, and thus there may always be an element of "mystery" about what is truly going on in the communications. This "mystery" is balanced by the fact the by adopting CORBA, the system integrators have left room to grow in the future in a number of different directions and can possibly leverage off of third-party software to build better systems in the first place.

Perhaps the weighting of the logistical criteria should rest upon the expected duration of the project. If the project is a short-term project with a set demonstration, then a small, properly chosen package such as IPT or RTC may be appropriate in order to quickly get the demonstration working. If the project is a long-term project with nebulous, possibly changing goals, then a more flexible and standard, but more complex and less predictable, communications infrastructure such as CORBA may be called for in order to weather the inevitable changes in project direction.

References

[1] J. Gowdy, "IPT: An object oriented toolkit for interprocess communication," Tech. Rep. CMU-RI-TR-96-07, Robotics Institute, Carnegie Mellon, March 1996.

[2] O. Firschein and T. M. Strat, eds., *Reconnaissance, Surveillance, and Target Acquisition for the Unmanned Ground Vehicle: Providing Surveillance "Eyes" for an*

Autonomous Vehicle. Morgan Kaufmann Publishers, 1997.

[3] J. Pedersen, "Robust communications for high bandwidth real-time systems," Tech. Rep. CMU-RI-TR-98-13, Carnegie Mellon University, 1998.

[4] J. Pedersen, "Real time communications." http://cronos.rec.ri.cmu.edu/technology/rtc/, 1998.

[5] National Institute for Standards and Technology, "The NML programmer's guide (C++ version)." http://www.isd.mel.nist.gov/projects/rcs_lib/NMLcpp.html, 1999.

[6] J. S. Albus, H. G. McCain, and R. Lumia, "NASA/NBS standard reference model for telerobot control," Tech. Rep. 1235, NBS, 1987.

[7] National Institute for Standards and Technology, "NML query/reply service." http://www.isd.mel.nist.gov/projects/rcs_lib/nmlqr.html, 1999.

[8] Real Time Innovations, Inc., Sunnyvale, California, *Network Data Delivery Service: The Real-Time Connectivity Solution*, 1.0 ed., June 1996.

[9] Real Time Innovations, Inc., "NDDS: Real-time networking made simple." http://www.rti.com/products/ndds/ndds.html, 2000.

[10] Message Passing Interface Forum, "MPI: A message-passing interface standard," Tech. Rep. CS-94-230, University of Tennessee, April 1994.

[11] Message Passing Interface Forum, "Message passing interface forum." http://www.mpi-forum.org/, 2000.

[12] G. Booch, *Object Oriented Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1991.

[13] The Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[14] Object Management Group, "OMG." http://www.omg.org/, 2000.

[15] Object Management Group, "Realtime CORBA joint revised submission," Tech. Rep. OMG Document orbos/99-02-12, Object Management Group, March 1999.

[16] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design and performance of real-time object request brokers," *Computer Communications*, vol. 21, pp. 294–163, April 1998.

[17] D. C. Schmidt, "Real time CORBA with TAO (the ACE ORB)." http://www.cs.wustl.edu/~schmidt/TAO.html, 2000.

[18] Object Computing, Inc., "The ACE ORB." http://www.theaceorb.com, 2000.