NetP: A Network API for Building Heterogeneous Modular Intelligent Systems

Kai-Yuh Hsiao and **Peter Gorniak** and **Deb Roy**
Cognitive Machines Group
MIT Media Laboratory
20 Ames St., Cambridge, MA, 02142
{eepness,pgorniak,dkroy}@media.mit.edu

Abstract

Any intelligent system, interacting with humans and acting in the real world, will be composed of many parts, each performing disparate and specialized tasks with wildly differing requirements. We have implemented a network abstraction layer, called NetP, that allows processes to communicate by sending structured data via non-blocking broadcasts to named channels. Any process can subscribe to any channel and receive the data posted to that channel. Using our API requires minimal programming effort, provides a simple network abstraction, and facilitates the debugging of network operations. The resulting system is resilient to partial failures and rapidly reconfigurable. In this paper, we present our networking approach, along with a case study that motivated our target feature list.

Introduction

In order to build an "intelligent" system, one that interacts naturally with human beings and acts competently in the real world, many different pieces, with many different requirements, must come together coherently. On some levels, such as when driving motors, such a system might need to work on a very fine-grained timescale and perform detailed real-time planning. On another level, the system might need to pass around large amounts of visual or auditory data for perceptual processing. On a third level, it might need to make and maintain structured beliefs about objects in the world, address hierarchical goals, and keep track of conversations.

In such a system, even if it only operates in a small niche of the real world, a large set of disparate pieces have to work together and communicate for the whole system to function. Clearly, modularity is the key to coherently integrating such a diverse set of abilities. However, the means of communication between modules strongly influences the level of complexity that can be feasibly attained. We have developed a network messaging paradigm with the goal of smoothly supporting a whole range of requirements, from high-bandwidth video data streaming to transmitting complex data structures.

Our messaging paradigm, named NetP, is centered around the concept of *broadcasting* to *channels*. Information is broadcast throughout the system, identified by the named channel on which it is transmitted. Any module can broadcast information to any channel. Likewise, any module can subscribe to any channel and receive any subsequent information sent to that channel.

Data sent across our system is required to be structured in a specific way, using map and list containers that hold integer, floating-point, and string primitive data elements. The constrained structure allows complex data types to be transmitted while eliminating dependence on user-defined message structures, making the system easier to maintain and more backwards-compatible. The data is sent as text, so all packets are transparently readable by all users for debugging purposes. For efficiency, we also provide routines for easily setting up a higher-bandwidth direct TCP/IP connection. Our API (the functions used to access the library) is geared towards simplicity for the programmer, and we have ported the API for use in C++, Python, and Java, running on Linux and OS X platforms.

Our complex systems rely on the features of our network paradigm. For example, one of our robots has a vision system, a motor system, a speech and language system, and a behavior system, each of which consist of multiple submodules. These modules have numerous authors, evolving specifications, and wildly different requirements with respect to accuracy of timing, computational power, and access to external resources such as cameras or motors. We also re-use modules across different systems, making it necessary for them to fit into different collections of modules at different times. With such large-scale interdependencies, care must be taken to limit the effects of any single change or defect. Our network system is designed to allow maximal functionality from such an unruly crowd of components with minimum time spent coordinating.

In this paper, we describe our network paradigm, along with properties of the paradigm that we find useful. We describe a case study showing how this strategy plays out for a real system. We conclude with our future plans for a large scale system for intelligent behavior using the same method.

Design and Implementation

In order to enable the smooth integration of large numbers of disparate modules, our group has implemented a network messaging paradigm, named NetP, that caters to a diverse set of networking requirements. Here we describe the basic approach, along with salient features and examples of how it is called from programs.

Broadcasting on Channels

Our modules communicate by broadcasting messages to named channels, to which other modules subscribe in order to receive the messages. A process accomplishes this by creating one or more instances of class `NetP`. Each instance of `NetP` can send on one channel, or receive on numerous channels. A channel exists as soon as any process subscribes to a channel for sending or receiving.

Channels represent an abstraction on top of the actual network. A process subscribed to a channel will receive all messages sent to that channel, regardless of where among the networked computers the process is running. We currently accomplish this by using a subset of functions provided by the PVM (Sunderam 1990) system, although the NetP system completely abstracts away this layer as well, so any back-end network setup that allows processes to join broadcast channels in an abstracted and distributed manner would work.

In order to make use of our system, all computers that will be involved in networked setup must be connected in a PVM network. A file can be made with the list of hosts in it, and the PVM network can be started on all hosts with a single command. Once this is done, all processes running on connected machines have access to the channel abstraction.

Our implementation of the channel broadcast abstraction provides us with several benefits:

Module separation Broadcasting messages between modules gives a clear sense of where one module ends and where another begins. Modules have access only to data sent via messages from other modules; they cannot arbitrarily call each others' functions and access each others' private data.

Module replacement Subscribing to named channels allows modules to listen for a message type regardless of the source, and send messages regardless of the destination. Substituting one module with another of similar functionality requires little more than subscribing or publishing to the appropriate channels.

Network abstraction As mentioned, processes can subscribe to channels without regard to where other processes are located on the network. If a machine is under heavy load, processes can be moved to other machines with no reconfiguration.

Network efficiency Thanks to the distributed PVM system, messages for a given channel are relayed over the network only to hosts with processes subscribed to that channel. Other machines do not see the message, and there is no central server or node through which all messages must pass. PVM accomplishes this by running its daemon on each machine and monitoring connected processes for channel subscriptions.

Threaded, Non-blocking Sending and Receiving

The NetP system's send and receive calls are non-blocking by default. For sending, this means that the sending process does not wait for any responses. For receiving, this means that receive calls return immediately, whether or not anything is waiting in the receive buffer, so the program can continue executing. This stands in contrast to RPC (Remote Procedure Call) methods, in which network operations are treated like function calls, which block while waiting for the other process to respond.

Naturally, this means that our system operates asynchronously, meaning that every module runs at its own pace and periodically checks its buffers for messages from other modules. This property makes the most sense for disparate sets of processes with vastly different timing requirements; it makes little sense for a real-time motor control process running at 1 KHz to run in lock-step with a vision system that only sees fifteen frames per second.

Within our non-blocking broadcast paradigm, we also provide the ability to perform blocking receives, for processes that need to wait for responses, and for sets of processes that need to synchronize. The NetP system runs in a separate thread (and measures have been taken to ensure thread safety), so the receive buffer can be managed independently of the main process, and so blocking receive operations can wake up the main thread when needed.

Using non-blocking operations via a separate thread offers several advantages:

Subsystem coherence Because most modules are not waiting for anything, they can run even when some of their messaging partners are not. Even if the vision system typically communicates with the mental model, the mental model can be debugged while the vision system is offline, by continuing to run while making use of any remaining inputs. Likewise, the vision system can produce outputs without expecting the mental model to receive them.

Start sequence insensitivity Modules can be brought up in any order, and when messages are available they will be processed. This eliminates the need for complicated scripting systems for bringing up processes according to complex dependencies.

Robustness to failure Individual modules can crash or be taken offline, and all the other modules will continue running at normal speed. While the entire system will not be complete with missing modules, the missing modules can be brought back online smoothly.

Buffer safety We manage receive buffers in a separate thread, so if a receive buffer is not read in a timely manner, it can be automatically truncated before occupying all available system memory.

Flexible synchronicity As mentioned, modules can opt to block on operations, so synchronous operation between tightly-related modules is made possible without forcing the whole system to adopt such a paradigm.

Structured Data and Text Representation

Data structures sent via NetP objects are packed into hierarchical data structures of type PData. PData is based on data structures in Python, allowing simple data elements of integer, floating-point, and string type, as well as complex data elements of list and map type that arbitrarily contain the simple data types. Access functions are provided so complex structures can be traversed easily. The top-level PData in each NetP instance is a map-type PData, encouraging messages to use labeled fields.

The purpose of creating a unique structured data type for NetP transmission is to eliminate the need for intermediate message classes. Without a standardized data type, the structure of each message type would have to specified separately, in an intermediate class shared between the sending and receiving processes so both ends can correctly interpret the structure of the data. This leads to versioning difficulties, as modules compiled with old versions of the message class would need to be constantly rebuilt to incorporate new versions. It also leads to an unmanageable proliferation of intermediate message classes.

We use map-type structures at the top level, encouraging the use of labeled fields. An example of this would be:

```
{
  'command':'goto_point',
  'joint':1,
  'position':0.53
}
```

Furthermore, all packets are sent over the network in text format, so every step of the way, the contents of a message are transparently human-readable for debugging purposes. Packets are accompanied by metadata, including the time of transmission, the hostname of the sending computer, and the process ID of the sending process. Timestamps are synchronized by running a program, `timesynchost`, on the computer whose clock should be treated as the central clock of the system.

We opted to use a Python-like representation instead of another text representation such as XML because Python representations can be directly coded in a rapid and intuitive manner by programmers. Using an XML-based representation could conceivably be useful if the system needed to be extended to support a much larger number of data types, in which case a translation layer from programmer-intuitive structures to XML sent over the network might be useful. We have not yet needed such extensibility, though.

The structured representation presents a few key features:

Transparently structured data Drawing from its Python inspiration, PData allows arbitrarily complex structures to be translated into a set of nested lists and maps. This standardization means that programmers writing receiving processes can easily read the format of a message and infer the appropriate routine for making sense of the message. This in turn reduces the amount of time that writers of modules need to spend coordinating their module communications.

Backwards compatibility When changes are made to message structure, new fields can be added under new labels, and old fields can retain their old labels. Thus, minor message format changes can still interoperate with old versions of receiving code. Likewise, new receiving processes can check the presence of new fields and respond accordingly. This stands in contrast to a system with a pre-specified raw byte messaging format, in which changes to message formats can lead to complete incoherence in message comprehension.

Accountability The metadata sent with each packet means that the origin of packets can be traced efficiently. This allows rapid identification of duplicate processes, as well as processes that are flooding the network. The simplicity of the channel subscription abstraction also allows logging facilities to gain access to all packets sent across the network, so the network as a whole can be observed and debugged.

Multiple Language and Platform Support

We have used the SWIG wrapper generator (Beazley 1996) to directly port the NetP and PData interfaces from C++ to Java and Python. The API looks essentially the same in all three languages, and in Python the translation to and from PData is even more simplified since PData representations have the same textual form as Python data structures.

Furthermore, we have successfully run the NetP system in both Linux and OS X. Although we have run PVM in a Microsoft Windows environment before, configuring PVM in Windows was frustrating and we have not attempted further porting to that platform.

Naturally, this provides one key benefit:

Uniform cross-language API Porting a single API allows programmers to learn one messaging standard, instead of (for instance) PVM in C++, jPVM in Java, and pypvm for Python, all of which have separate setup and usage procedures. API changes in one language are immediately reflected in the API's of the other languages.

It is our belief that our API is extremely easy for programmers to use, and we intend it this way so even novice programmers can integrate modules without much effort. In Figures 1, 2, and 3, we provide simple examples of sending and receiving in each of the three languages. The packet being sent across the network in all three examples is:

```
#{'num': 1, 'list': [1, 2, 3, 4, 5]}#.
```

Direct Connections

For most purposes, the NetP system provides a reasonable tradeoff between transparency and raw speed. Without the parsing of structured packets sent in text format, and without using PVM's groups (channel) facilities, the system would run slightly faster. However, this increase in speed is undesirable for typical messaging needs, because the time spent debugging and maintaining would increase to a frustrating extent. Thus, for most process-to-process communications, we adhere to the structured, buffered communications provided by our current system.

However, for the processes with extremely high throughput, e.g. raw video streaming, it might be wiser to forego

```
#include "NetP.h"
int main() {
  NetP sendp("MyChannel"); // argument to constructor means it’s a send instance
  sendp.data()["num"] = 1; // make a field named "num" and put 1 in it
  sendp.data()["list"].eval("[1, 2, 3, 4, 5]"); // make a field "list" and eval a
     // string representation of a list into it
  sendp.send();
}
--------
#include "NetP.h"
int main() {
  NetP recvp; // no constructor argument means it’s a receive instance
  recvp.joinChannel("MyChannel");
  while (1) { // loop forever
    string resp = recvp.bReceive(); // blocking receive
    int value = (int)recvp.data()["num"]; // get an int value
    int listval = (int)recvp.data()["list"][3]; // get a value from a list
  }
```

Figure 1: C++ send and receive code examples.

```
import netpacket
sendp = netpacket.NetP("MyChannel") # argument means it’s a send instance
a = {’num’:1, ’list’=[1, 2, 3, 4, 5]}
sendp.data().eval(repr(a)) # copy the structure into the NetP
sendp.send()
--------
import netpacket
recvp = netpacket.NetP() # no argument means it’s a receive instance
while (True):
    resp = recvp.bReceive() # blocking receive
    a = eval(repr(recvp)) # _a_ now holds the structure
```

Figure 2: Python send and receive code examples.

```
import netpacket.NetP;
class examplesend {
  public static void main(String[] args) {
    NetP sendp = new NetP("MyChannel"); // channel argument means it’s a send instance
    sendp.data().get("num").set(1); // make a field named "num" and put 1 in it
    sendp.data().get("list").eval("[1, 2, 3, 4, 5]"); // make a field "list" and eval
      // a string representation of a list into it
    sendp.send();
  }
}
--------
import netpacket.NetP;
class examplerecv {
  public static void main(String[] args) {
    NetP recvp = new NetP(); // no argument means it’s a receive instance
    recvp.joinChannel("MyChannel");
    while (true) {
      String resp = recvp.bReceive(); // blocking receive
      int value = recvp.data().get("num").getInt(); // get an int value
      int listval = recvp.data().get("list").get(3).getInt(); // get a value from list
```

Figure 3: Java send and receive code examples.

```
#include "NetP.h"
int main() {
  NetP::DirectOffer("DirectTest"); // prepare to receive connection; does not block
  while (NetP::GetSocketNum("DirectTest") == -1) { // sleep until connection exists
    sleep(1); // you could do other stuff here too
  }
  for (int i=0; i<10; i++) {
    cout << "read " << NetP::DirectRead("DirectTest") << endl;
    sleep(1);
    cout << "write " << NetP::DirectWrite("DirectTest", "qwertyuiop") << endl;
    sleep(1);
    if (NetP::GetSocketNum("DirectTest") == -1) { // if connection disappears then stop
      break;
    }
  }
}
--------
#include "NetP.h"
int main() {
  NetP::DirectConnect("DirectTest"); // initiate connection; blocks until something receives
  for (int i=0; i<10; i++) {
    cout << "write " << NetP::DirectWrite("DirectTest", "asdfjkl;") << endl;
    sleep(1);
    cout << "read " << NetP::DirectRead("DirectTest") << endl;
    sleep(1);
    if (NetP::GetSocketNum("DirectTest") == -1) { // if connection disappears then stop
      break;
    }
  }
}
```

Figure 4: Code example for setting up a direct TCP/IP connection.

structured data and the additional buffering that PVM requires. To this end, there is also the possibility of forming a direct TCP/IP connection between two processes. This direct connection is made very simple to use by abstracting it via the NetP architecture. One process runs the DirectOffer() function, specifying a channel name, and another process runs the DirectConnect() function, specifying the same channel name. As long as exactly one process has run each function for a given channel name, regardless of order, the NetP system will negotiate the connection, form the direct TCP/IP connection, and provide a raw read and write function for raw binary data transfer.

See Figure 4 for an example of setting up a direct connection. Naturally, the DirectConnect grants one additional benefit to our system:

Enabling speed/structure tradeoff Most processes will send structured messages with varying levels of structure and hierarchy. However, for processes that require extremely high throughput, raw minimally-buffered binary data transfer can be used without the overhead of channel abstractions, structure parsing, and textual representation.

A Motivating Case Study

The feature requirements that motivated our current network approach are the result of our experiences developing a large-scale intelligent robotic platform using a preliminary form of our network paradigm. Our preliminary approach made direct calls to PVM, which provided access to the benefits of channel-like abstraction across the network.

However, directly calling PVM involved transmitting raw byte sequences instead of structured representations, and omitting many convenience features we have since added into our complete abstraction layer. In order to make sense of the raw byte sequences in each message, we implemented a separate class for each message type that needed to be sent. These message classes each contained the necessary procedures for generating raw messages and extracting structure and data fields from raw messages. Thus, each message class needed to be included in all modules that sent or received that message type.

Our primary system using this initial networking approach was a robotic platform, Ripley, whose pieces came together from a number of other separate projects over time. By standardizing all members of our group, even people working on unrelated projects, on our network paradigm, it became possible to efficiently bring together modules that were never originally intended to interact. This was accomplished by adding some message types and altering channel subscriptions appropriately.

Ripley (a fairly recent version is described in detail in (Roy, Hsiao, & Mavridis 2004)) is a robotic system that integrates motor, visual, speech, and language processing. It can accept commands such as "Hand me the blue object on my left," which make reference to positions of objects in its tabletop environment as well as to the user's position. Speech recognition is performed by the Sphinx-4 speech recognizer (Carnegie Mellon University, Sun Microsystems Laboratories, Mitsubishi Electric Research Laboratories 2004), which passes output to a language parsing system. Information about objects and the user is determined based on visual input and stored in a three-dimensional mental model. The language parser draws from the mental model to determine word referents, and then either chooses to ask simple questions (e.g. "Do you mean this one, or this one?") or to send commands to the motor system for execution.

For a simplified diagram of modules in the Ripley system, see Figure 5. None of these subsystems (speech, language, motor, visual, and mental model) were initially designed for the purpose of integration into the final version of the Ripley system. The system most closely tied to the Ripley platform, the motor system, simply broadcasts state information over a specific channel to any process that wishes to listen, and listens for command messages.

The current speech recognition system is built around the Sphinx-4 speech recognizer by using a wrapper that broadcasts recognizer output over the network. We are using the same wrapped speech recognizer module in numerous projects simultaneously, by setting up separate PVM networks and having other modules listen to the speech recognizer's output channel.

The language parsing system was initially part of the Bishop system (Gorniak & Roy 2004), which parsed speech input from the recognition module to find referents in a simulated world of purple and green cones. Similarly, the visual system was initially part of the Newt robot platform (Roy *et al.* 2002), which used our own speech input system to learn how to attach words to the appropriate visual features of objects found in the camera input. Using the network paradigm introduced here, Sphinx 4 later easily replaced our own speech recognizer by simply emulating its channels and messages.

The first approach to integration involved just the motor control system and the visual modules from the Newt system. The mental model was constructed (Hsiao, Mavridis, & Roy 2003) using the vision system as input and the motor system as output. As mentioned, integration of a new module consisted basically of setting up each module to listen to the channel broadcasts that contained the necessary information. Our initial system made use of only primitive phrase-spotting to carry out simple commands.

Later on, it became evident that the language parsing and referent selection of the Bishop system would be suitable as a language module for the Ripley system. Once again, the basic network integration involved setting up approriate message types and listening to the right channels. Pieces of the Bishop system were integrated to enable language processing on the speech input and visual processing on the current contents of the mental model. In essence, we were able to replace Bishop's synthetic vision system with Ripley's real one using the same set of channels and message types. Naturally, additional effort was required to reshape the referent-finding modules to coherently handle the specific representations of the mental model (i.e. varieties of objects beyond green and purple cones).

The construction of the complete Ripley system undeniably involved a lot of effort and complexity, but we found the process to be greatly simplified by being able to subscribe to broadcast channels with evolving message types instead of having to configure specific host/process connections. The nonblocking nature of our network calls also simplified debugging because subsections of the system could run coherently in the absence of others (e.g. we could monitor visual processing without requiring speech or motor control to be active).

Lessons Learned

However, we also ran into difficulties with this approach, which became the inspiration for many of the features in our new system. The number of message types ballooned rapidly and it became difficult to track down specifications for each message type and which modules it belonged to. Disparity in versions of message types became a headache of recompilation, and much time was spent reconciling message specifications. Our structured data format in the NetP system is designed to avoid this scalability problem.

Also, PVM places all received messages in subscribed groups into a buffer, but the receiving process is responsible for reading the messages in a timely manner. Not all processes were so carefully written, and this resulted in buffers growing without bounds, and computers running out of memory and crashing. To address these, our new system uses its separate buffer-management thread.

Furthermore, with most message wrappers written in C++, occasionally implementing modules in Java (with jPVM) and Python (with pypvm) meant reimplementing each message type for each language, which led to more version disparities.

Other Possible Solutions

After looking for a networking setup for our modular designs, we settled on our custom library based around structured channel broadcasts. In this section we overview other approaches to network integration.

One approach to this problem is to pick an existing high level language paradigm for distributed applications. These include RPC (Remote Procedure Call, with implementations like ONC-RPC (Srinivasan 1995), XML-RPC (Winer 1999), CORBA (Object Management Group 2004), and SOAP (XML Protocol Working Group 2003)). Another similar method is RMI (Remote Method Invocation), with implementations like Java RMI (Sun Microsystems 2003)). These allow processes to call functions and receive responses over a remote network connection. While these approaches cer-

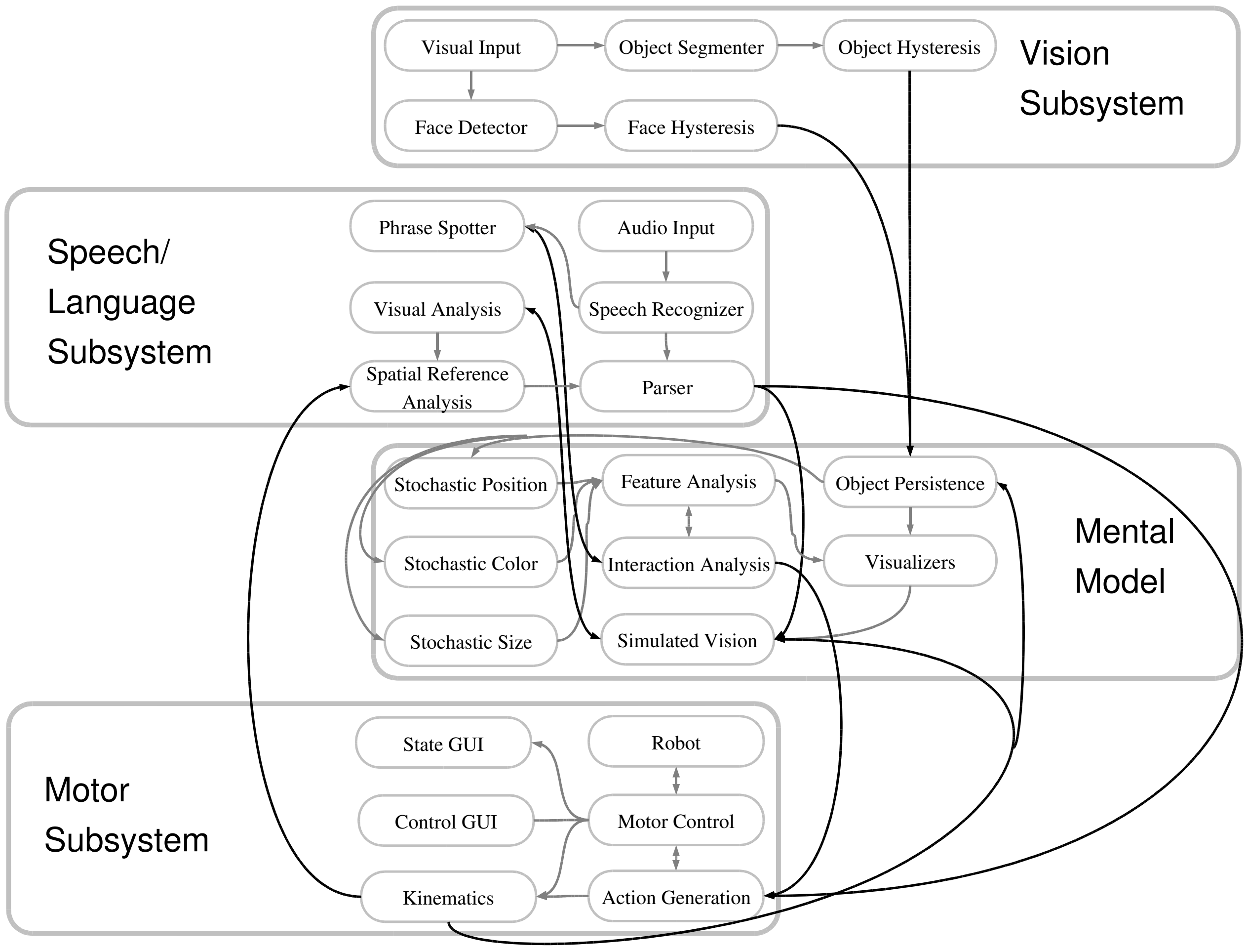

Figure 5: Diagram of Ripley modules and their messaging relationships.

tainly see widespread support for multiple programming languages and operating systems, they are typically used for synchronized operation of pairs of processes, and require each network connection to be made to a known machine. A function call is an inherently blocking operation, so broadcasts and asynchronous use is nonnative to these paradigms. Transparency, monitoring, and accountability are also limited with these systems. Most importantly, these approaches rely on strictly-defined interfaces and availability of the connected processes.

Low level message passing interfaces such as PVM (Parallel Virtual Machine (Sunderam 1990)) or MPI (Message Passing Interface (MPI Forum 1997)), on their own, also fail to meet our requirements. They send fixed-structure raw messages that lack meta-level data and are not convenient for complex data structures. Their logging facilities are at such a low level that decoding and abstracting them into meaningful information is a large effort. Finally, their threading and buffer management facilities are limited, often forcing the developer to write his or her own.

Another possible approach involves multicasting over TCP/IP networks (Deering 1989). This enables packets to be sent to broadcast addresses, and any host listening on a particular address can receive the relevant packets. This could conceivably produce the same channel-listening behavior as our current system. Our emphasis in producing NetP is to provide the simplest multi-platform API that we can to collaborators, regardless of the underlying technology. Multicasting could well be a viable replacement for our PVM back-end, although multicast does encounter issues with kernel configuration and router compatibility, and these would have to be examined and overcome.

Every intelligent system adopts its own networking strategy. As an example, Cohen et al.'s agent architecture (Cohen *et al.* 1994) makes use of a networked blackboard configuration where agents post problems to a hierarchical set of blackboards, and other agents solve the problems and post the solutions. All communication is handled by a central set of blackboard servers. Other blackboard methodologies are discussed in (Isla & Blumberg 2002).

Our NetP architecture is designed for networked communication in a more distributed fashion, instead of designating a blackboard server or servers. Furthermore, many blackboard systems are designed for specific types of agent-to-agent communication. The NetP system does not preclude sending to named channels whose receivers implement a blackboard architecture. Rather, NetP enables rapid setup of networking for exactly this sort of configuration, and at the same time enables other processes with completely different needs to transmit data in formats appropriate to them.

Future Plans

Our new system is now in place and being used for a number of small projects, each involving several modules. In the Ripley system, the speech, language, and motor systems have mostly been transitioned to the new abstraction layer, although some sections are still using the legacy framework. Because of the improvements in scalability and transparency, debug and maintenance time for the updated modules has become less frustrating.

A new robot is currently under construction in our group, and it promises to require more complexity and integration than our previous robot projects. By having NetP up and running before the modules of the new robot come online, we hope to minimize network integration time and make it easier for more participants to add functionality to the new robot. NetP has already been incorporated into the hand grip controller for the new robot. Efforts have also started to build an overall cognitive scheme for the new robot, in which all modules will be further constrained to a specific protocol so that multiple modules can produce conflicting outputs and resolve the conflicts smoothly.

Work is also ongoing to add features to the NetP architecture. We are preparing to release the NetP system under the open source LGPL license to enable anyone who is in need of a network system with similar functionality to share and contribute. This will be located on the web at *http://netp.sf.net*.

Conclusion

We have developed a networking paradigm, and implemented an abstraction layer and an API, called NetP, for networked communication of a system made up of many heterogeneous processes. By using nonblocking broadcasts within a channel subscription model, processes can be rapidly reconfigured to send and receive from one another, to require no reconfiguration when hosts and processes are moved around the network, and to continue operating when related processes are offline. We require communications to use a uniformly structured data type and transmit it with metadata, which enables all packets sent over the system to be human-readable for debugging and accountability. This also allows message specifications to change while still being backwards-compatible. The network system is also designed to run in a separate thread so it can perform buffer management in the background and provide blocking and synchronization to processes that request it. Finally, we have used wrappers to port the C++ API directly to Java and Python, and run the system on both Linux and OS X machines to provide implementation flexibility to the writers of modules.

References

Beazley, D. M. 1996. Swig: an easy to use tool for integrating scripting languages with c and c++. http://www.swig.org/papers/Tcl96/tcl96.html.

Cohen, P. R.; Cheyer, A.; Wang, M.; and Baeg, S. C. 1994. An open agent architecture. In *AAAI Spring Symposium*, 1–8.

Carnegie Mellon University, Sun Microsystems Laboratories, Mitsubishi Electric Research Laboratories. 2004. Sphinx 4 Java Speech Recognizer. http://www.speech.cs.cmu.edu/cgi-bin/cmusphinx/twiki/view/Sphinx4/WebHome.

Deering, S. 1989. Host extensions for ip multicasting. http://www.ietf.org/rfc/rfc1112.txt.

Gorniak, P., and Roy, D. 2004. Grounded semantic composition for visual scenes. *Journal of Artificial Intelligence Research* 21:429–470.

Hsiao, K.; Mavridis, N.; and Roy, D. 2003. Coupling perception and simulation: steps towards conversational robotics. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 928–933.

Isla, D., and Blumberg, B. 2002. Blackboard architectures. In Rabin, S., ed., *AI Game Programming Wisdom*. Charles River Media, Inc. 333–344.

MPI Forum. 1997. Mpi-2 standard specification. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

Object Management Group. 2004. Common object request broker architecture: core specification. http://www.omg.org/technology/documents/formal /corba_iiop.htm.

Sun Microsystems. 2003. Java RMI specification. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/index.html.

XML Protocol Working Group. 2003. Soap version 1.2 specification. http://www.w3.org/TR/soap/.

Roy, D.; Gorniak, P.; Mukherjee, N.; and Juster, J. 2002. A trainable spoken language understanding system for visual object selection. In *International conference of spoken language processing*.

Roy, D.; Hsiao, K.; and Mavridis, N. 2004. Mental imagery for a conversational robot. *IEEE Transactions on Systems, Man, and Cybernetics* 34:1374–1383.

Srinivasan, R. 1995. Rpc: Remote procedure call protocol specification version 2. http://www.ietf.org/rfc/rfc1831.txt.

Sunderam, V. S. 1990. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience* 2(4):315–339.

Winer, D. 1999. Xml-rpc specification. http://www.xmlrpc.com/spec.