

Supporting Robot Teams with CougaarME over Wireless Ad-hoc Networks

Josh Bers and Jason Redi

BBN Technologies

jbers@bbn.com

Abstract

A Cougaar-based software architecture is developed and deployed on a team of mobile robots. The resulting system leverages Cougaar Micro Edition's (ME) tasking framework and its Message Transport Service for task delegation and reporting across a wireless ad-hoc network. CougaarME also supports seamless information sharing between the mission control and ad-hoc networking modules via its blackboard service. We briefly review existing robot systems to motivate our selection of CougaarME. We describe the robotic hardware and software platform. Then we present the software architecture, including enhancements made to CougaarME and its multimodal user interface. Finally we offer some lessons learned from developing an embedded Cougaar application over a mobile ad-hoc network.

1. Introduction

This paper describes the implementation of a distributed multi-robot system. The purpose of the system is to provide a robotic platform that enables interactions both intra-robot, between movement control algorithms and the wireless network, and inter-robot between semi-autonomous robots as well as with a human operator. Our architecture uses CougaarME as the collaboration framework. We extend existing classes to provide task-based access to information from the mobile ad-hoc network and control of the robot hardware. Each robot contains a single CougaarME agent with plugins for mission, mobile network and robot control all collaborating via the Blackboard service. The Message Transport Service provides inter-robot tasking and control from a remote user-interface. Our task language defines two-levels of commands to accommodate the two types of interactions: local and remote.

Section 2 provides some background on related systems including our reasons for selecting CougaarME as our collaboration solution. Section 3 describes our robotic platform's hardware and software. Section 4 presents the details of the software architecture. Section 5 describes limitations and suggests future work. Finally section 6 summarizes our contribution.

2. Motivation and related work

The primary goal of this work was to build a robotic testbed to support research on how distributed autonomous robotic behaviors can be enhanced through knowledge of the wireless ad-hoc network that connects them and visa-versa. This could help to answer questions like: Does network-awareness help mission planning algorithms and/or remote operators to make complex decisions? e.g., in a communications relay network, which nodes should move to improve global connectivity? We have described results of this research elsewhere [1, 2]. The focus of this paper is on the design and implementation of our robotic testbed.

In order to support this goal we developed seven requirements for our ideal collaboration framework: R1: freely available, R2: transparent data sharing, R3: distributed software environment, R4: support parallel distributed processing for any component, R5: promote ease of development, R6: platform independent, and R7: lightweight 0. In the next section we review existing systems in light of our seven requirements and then describe our selection.

2.1. Collaborative architectures for autonomous robots

We found that while there is much work on autonomous robots, there is no agreed upon standard for a high-level software framework, and thus much duplication of effort. There are many proprietary systems developed for specific hardware and targeted applications. However, there is no open standard for high-level robot control architecture.

There is, however, more agreement in the lower-layers of the robot control architecture. Many groups share a common functional layer that abstracts away from any specific robotic platform. For example, the open-source Player/Stage system originally developed at USC has a broad user-base within the robotics community 0. It provides a nice abstraction of the robot hardware and offers various device interfaces useful primarily at the reactive feedback level of robot control architecture.

We provide brief descriptions of some representative work in robot architectures.

Gat's "Three-layer Architectures," provides a framework for control processes at three different timescales: a reactive feedback control mechanism, a reactive plan execution mechanism and a mechanism for deliberative computations [1]. By contrast, the CLARAty architecture from JPL divides a robot system into two-layers, the decision layer and the functional layer [2]. The functional layer sits on top of the hardware and provides resource abstractions up to the decision layer.

Brugali and Fayad describe distributed architecture models: client-server, three tier, broker-based and multiagent [3]. Woo describes a CORBA-based robot application infrastructure, however, it is not designed to run in embedded, lightweight CPU's, nor across a wireless network [4].

2.2. Our hybrid architecture

The distributed multi-agent CougaarME architecture was chosen as our collaboration platform based on our requirements, since it: is freely available [R1], supports distributed parallel modules [R3, R4], and has a publish and subscribe data sharing mechanism [R2], and has been previously applied to an embedded robots application [R7]. Prior use of CougaarME for robots involved a full Cougaar node tasking many CougaarME nodes each running on separate CPUs all within a single robot platform [8]. By contrast we use a single CPU per robot running a single MicroAgent. A MicroAgent represents a single agent running within a single JVM, also known as a CougaarME node. The plugins that make up the MicroAgent use SNMP to communicate with the mobile ad-hoc network subsystem, PSF (portable switch framework), and use a Java client to interface to the Player robot control server [9]. Figure 1 shows the layering of our robot software architecture.

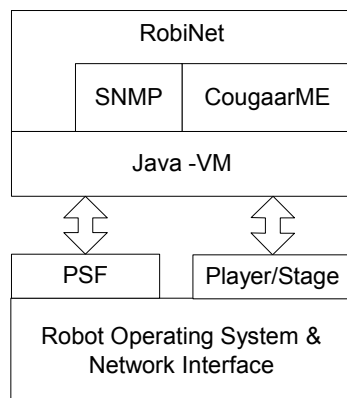


Figure 1. Layered software architecture.

In the next section we describe our robotic hardware platform.

3. Robotic platform

Our hardware platform includes ActivMedia AmigoBot robots, ARM processors for on-board robot control and x86 laptops for remote control, all of which run the Linux operating system. For networking we use wireless LAN PC cards running a customized proactive link-state ad-hoc routing protocol (MANET) with an enhanced SNMP status interface. In this section we describe the robotic hardware and software runtime platform.

3.1. Hardware



Figure 2. AmigoBot with Nano-engine and WiFi card.

We use the AmigoBot robot from ActivMedia [10]. Each robot is equipped with either a Nano-engine from Bright-Star engineering or an iPaQ PocketPC's attached to the robot chassis that controls the robot via an RS232 serial cable. For inter-robot communications we use a wireless Ethernet WLAN card. Figure 2 shows one of our robots.

The 1"x2" Nano-engine jointly designed by Navy SPAWAR and BBN, comes with a 200 MHz SA-1100 ARM processor, 4 MB flash and 32 MB RAM. The manufacturer, BSE, also provides a RIB-board daughter-card that accepts three serial connections, one type II PCMCIA card and an Ethernet interface. Newer versions come with 8 MB of flash and two PCMCIA slots. The iPaQ's typically come with 32 MB of flash and 32 MB of RAM.

We use Orinoco Silver wireless Ethernet cards from Proxim. They support the 802.11b protocol at data rates up to 11 Mbps. Their maximum effective data throughput is about 5 Mbps. More recently we have added support for Cisco Aironet cards.

3.1.1. Power Requirements. The iPaQ has its own battery, whereas the nano-engine gets its power from the robot battery. The most recent version of the nano-engine and RIB boards require 5 volts RMS, and draw an average current of 420 milliamps with the WiFi card active but not transmitting. On some robots we run a

webcam that draws an additional 700 milliamps. To accommodate the extra drain on the robot battery we added a second 12-volt battery in parallel with the standard battery. In this configuration, a fully loaded robot can run ~45 minutes between re-charges.

3.2. Software

As shown in Figure 1, our robotic framework is built on top of the Linux OS, the portable switch framework (PSF), the Player robot server, and the Java Runtime Environment. We have described the details of our ad-hoc routing system, PSF, elsewhere [1].

3.2.1. Linux kernel configuration. We run the Linux OS on both the iPaQ and the nano-engine. We run the familiar Linux distribution on the iPaQ. The process of installing the bootloader and familiar distribution is well documented at <http://www.handhelds.org/familiar>. The Nano engines come with a set of standard Linux software tools from BSE.

To build all of the component applications for either platform we use the GNU ARM cross-compiler found at: <http://handhelds.org/download/toolchain/>.

Due to the limited flash space on the nano-engine, a compressed kernel and filesystem are loaded into ram at boot-time. Components that are too large to store in flash are downloaded across the wireless network at boot time. On the iPaQ, however, once the bootloader and initial configuration has been installed updating the system is just a matter of copying files to the flash ram system. This can be done over the wireless network using scp.

3.2.2. Player. Although the AmigoBot comes with libraries for developing robot control software, we selected Player in part because of its widespread use by the robotics research community. Player provides a device independent robot control interface [R5] in a small flash footprint: ~200KB.

We began with version 1.2.4 of the Player server, which did not have support for the ARM processor. We debugged issues with the alignment of the sonar data coming from the p2os driver. We also added support for playing sound on the AmigoBot. Our fixes and additions appear in Player versions 1.3.2 and higher.

Details on configuring the Player release to cross compile for the ARM appear here: http://sourceforge.net/mailarchive/message.php?msg_id=2423056

3.2.3. Java JVM. We selected the Kaffe JVM since it provides a good compromise on speed, space and functionality. It gives us the functionality we need: J2SE 1.2, including floats, and the java.utils package. It takes up about 1.2 megabytes of the flash ram. When running,

the class library expands to 1.9 megabytes on our ramdisk. Kaffe also runs faster than the other option, Blackdown, because it uses a Just In Time compiler (JIT).

In order to run our GUI and speech interface from the iPaQ's we use Blackdown because it supports the full J2SE implementation including audio system and Swing support [11]. Since the distribution takes up too much space to keep it all on flash ram, 17 MB, we download it into a ramdisk at boot time.

Helpful information on Java for ARM Linux can be found on the handhelds.org website: <http://www.handhelds.org/z/wiki/JavaOnIPAQ>

3.2.4. Memory footprint. Our final implementation runs in about 12MB of RAM on the nano/iPaQ. The memory usage by process, as reported by the VmRSS field the in /proc/<pid>/status, is shown in Table 1.

Table 1. Memory usage.

<i>Process</i>	<i>Resident RAM</i>
Java (JVM+CougaarME+System)	7.5 MB
Player	1 MB
NetMgmt (PSF agent)	1.7 MB
UCD SNMP agent	1.5 MB

3.3. Platform lessons learned

- Beware drops in battery charge. They can wreck havoc on the system, typically the first devices to stop functioning are the PCMCIA devices and hence the WiFi card shuts down.
- Flash RAM can be slow, especially for writing. Use a ramdisk for fast runtime file-access.
- Use the wireless network to bootstrap large applications. We would keep some of the Java system compressed on flash and other Java class files stored on a remote laptop. At bootup time, the nano-engine would uncompress Java and then fetch and execute an installation script from the laptop. This allowed us to change to the robot software without having to re-flash the nano-engines each time.
- Beware of unseen WiFi access points and radios. A normally functioning WiFi card will suddenly switch into a channel-hopping mode where the ESSID and WiFi frequency changes rapidly causing loss of connectivity. We believe that access points that are configured to accept any ESSID cause this.

In the next section we describe our software architecture for our robotic testbed.

4. A lightweight agent architecture for distributed robots

The basis for collaboration and distribution in our architecture are Cougaar's shared Blackboard Service and its tasking framework. Both of these are well described in the Cougaar Developer's Guide and in the Cougaar Architecture Guide 0. Unlike its bigger brother, CougaarME's small footprint and its dependence on the J2ME language subset enables it to run on lightweight Java VM's such as the J2ME [13].

In the following sub-sections we describe enhancements made to CougaarME infrastructure for our environment, then our software implementation and finally the user interface.

4.1. Enhancements to CougaarME

The CougaarME framework facilitates distributed robot control through a Message Transport Service (MTS). This service connects MicroAgents together through a task distribution and synchronization mechanism [8].

The default transport mechanism used by the MTS is deficient for our needs, where robustness to connection failures is critical. The default service uses TCP sockets to send tasks as XML encoded messages between MicroAgents. We discovered that connection and transmission delays due to degraded wireless links could cause long interrupts in the operation of the graphical user interface. This was due to the GUI being locked out from accessing the shared blackboard while the MTS attempted to send a message. To alleviate this, we developed an asynchronous message transport that uses a pool of threads to perform the sending in the background outside of a blackboard transaction, allowing other plugins to continue using the blackboard in the face of network failures and timeouts.

Our message transport also uses UDP to transmit task updates or allocation results. This helps to ensure low latency for status reports while preventing occasional dropouts from causing major network delays due to TCP retransmissions. New tasks are still sent reliably, however, to ensure acknowledged delivery. Future work should develop reliable transport services that are wireless network friendly 0.

Our two-tiered control language enables robots to maintain direct control over their movements via publication of local BotMotion tasks by their MissionPlugins, while enabling indirect remote control via Remote-BotMotion tasks. Figure 3 shows the logical

connections between the control and robot nodes. Remote tasks are received by each robot's MissionPlugin and translated into local tasks. For example, the local MissionPlugin could expand a remote BotMotion:goto task into a sequence of BotMotion subtasks. Reporting on tasks is also intermediated by the local MissionPlugin. The robot MissionPlugin may choose to monitor all changes reported from the local task, whereas it may report only the significant changes on a remote-task. Aside from lowering bandwidth requirements on the network infrastructure and reducing information overload in the tasking, two-tiered tasking affords mixed-initiative behavioral control: local and remote.

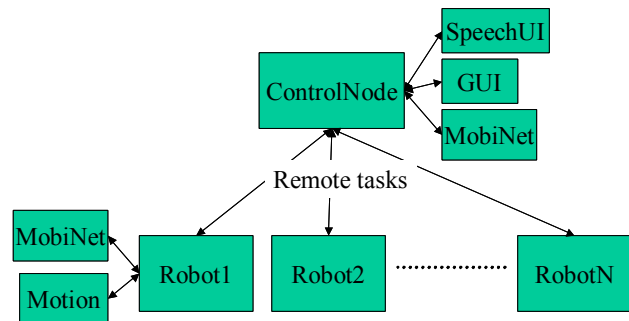


Figure 3. Two tiered control structure. Each robot controls its local resources, whereas control nodes can also remotely task robots. The MobiNet and Motion boxes represent the mobile ad hoc network and robot motion resources respectively. The GUI and SpeechUI represent the user interface resources on control nodes.

4.2. Software Implementation

The on-robot agent is made up of three plugins, MissionControl, MobiNetControl and RobotMotionControl, and two resources, MobiNetResource and RobotMotionResource.

The MissionControl receives remote-BotMotion tasks and translates them into local tasks. The MobiNetControl receives NetWatch tasks and sets up monitoring of the requested network information, e.g., link status. The RobotMotionControl receives local BotMotion tasks and creates and executes the corresponding behavior. It reports back on its progress, e.g., goto x, y.

Figure 4 shows the plugins that compose a CougaarME agent running on a robot. Table 2 shows the tasks that define the control language for the system. Figure 5 shows the UML class diagram for the robot agent. Figure 6 shows the sequence diagram for a position update going from the robot to the mobile network.

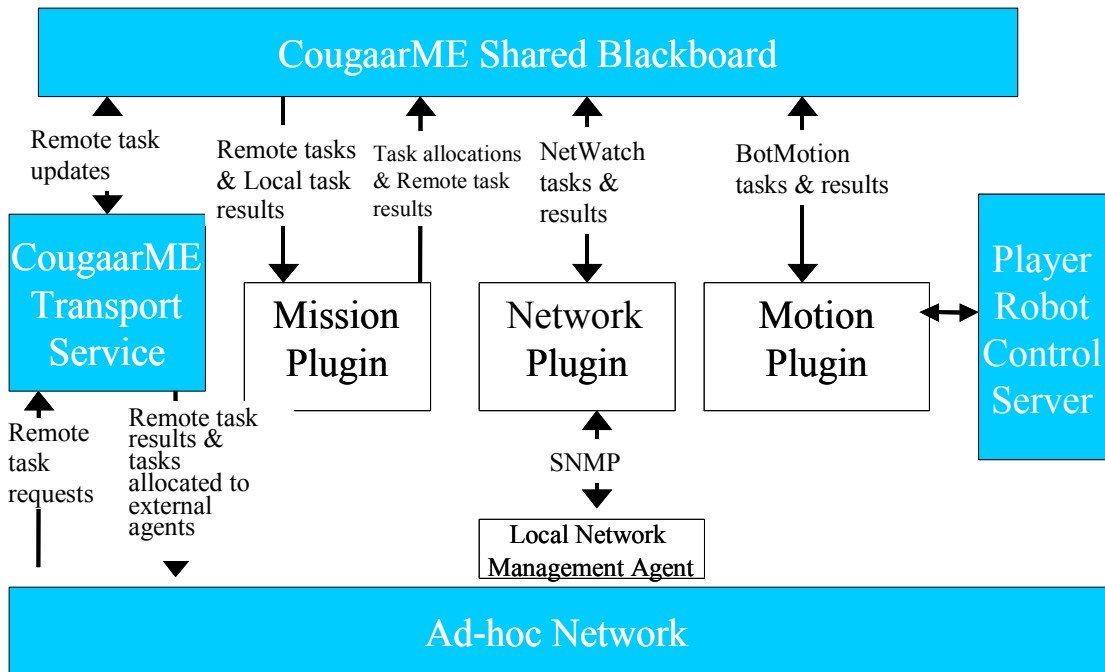


Figure 4. Block diagram showing the components of a CougaarME robot node and their logical connections.

Table 2. Task language for robot control.

<i>Verb</i>	NetWatch	BotMotion	Remote-BotMotion
<i>Semantics</i>	Subscribe to information about the network	Control low-level motion behaviors	Task remote robot movement
<i>Prepositions (subtask)</i>	Node position, network topology, path cost, link quality, neighbor list	Randomwalk, goto, stop, turn, teleport, clear, velocity, avoid obstacles, forward	Randomwalk, goto, cancel, teleport, forward
<i>Scope</i>	Local (intra-robot)	Local (intra-robot)	Team (inter-robot)
<i>Responsible Plugin</i>	MobiNetControl	RobotMotionControl	MissionControl

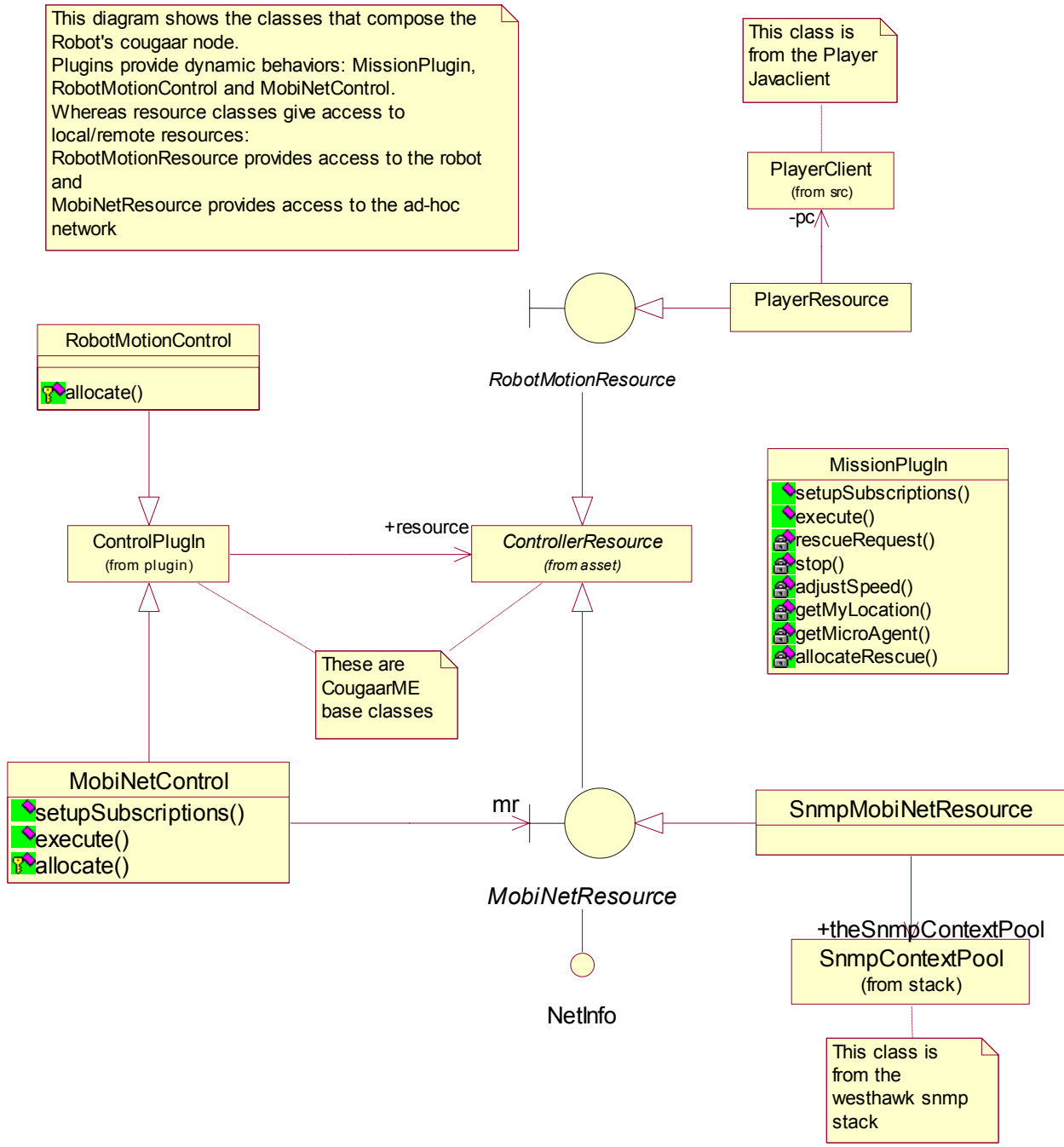


Figure 5. Class diagram for robot node.

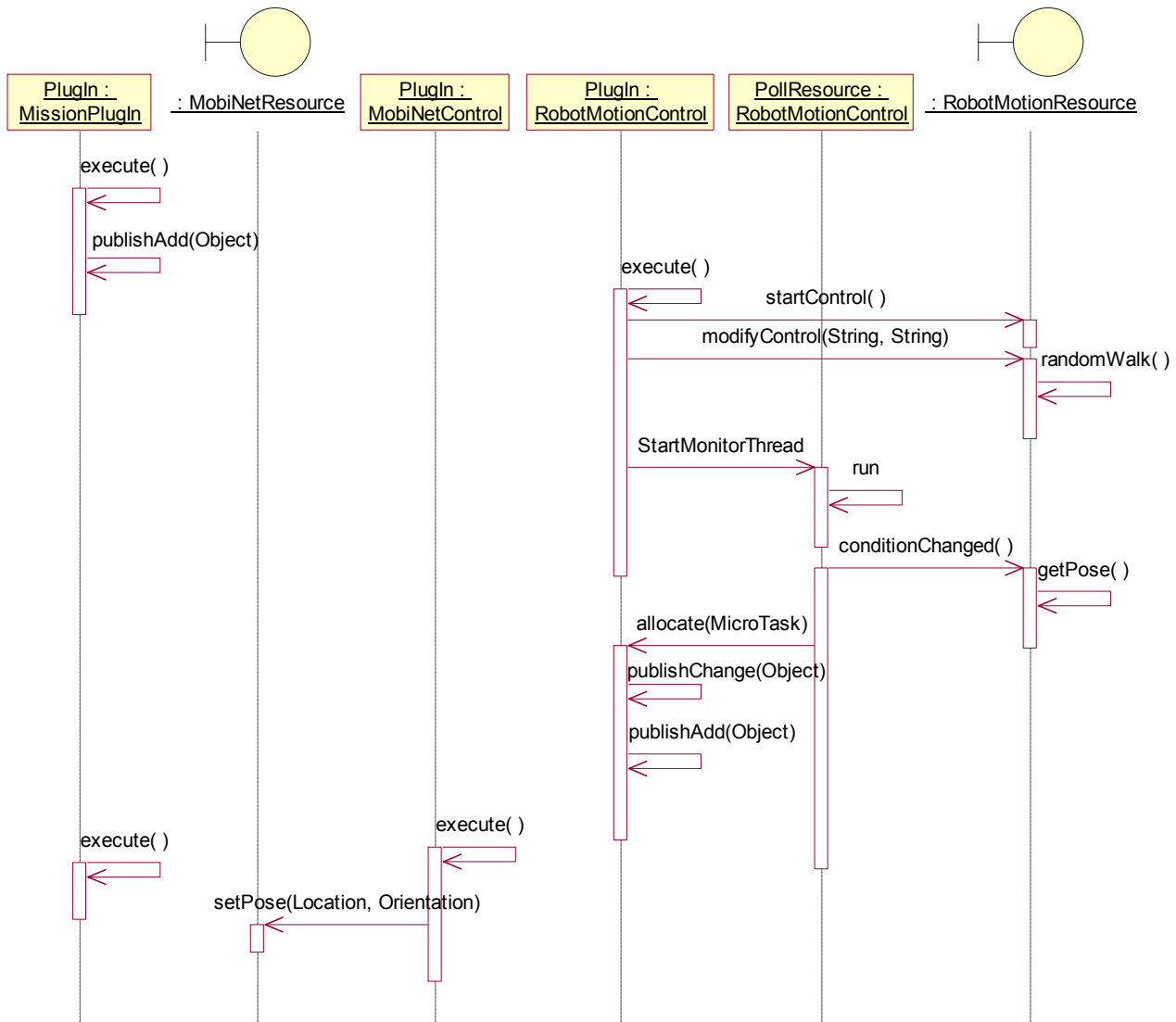


Figure 6. Sequence diagram showing interactions between plugins.

4.2.1. Task flow. A typical task flow and plugin interaction is illustrated in Figure 6: 1) MissionPlugin publishes a BotMotion:randomwalk task; 2) RobotMotionControl's task subscription fires; 3) It uses the RobotMotionResource to execute the task; 4) It also creates the PollResource thread to periodically check for changes in resource status; 5) As the robot moves, the PollResource thread publishes task allocation results as well as Pose objects to the blackboard; 6) MobiNetControl's Pose subscription fires causing it to tell the network about its new position via the MobiNetResource setPose method; 7) Finally the MissionPlugin's task subscription fires with AllocationResult updates to its original BotMotion task.

4.2.2. Simple reactive robot behaviors. We implement robot motion control via a stack of reactive behaviors. The RobotMotionResource class creates a separate Java Thread to service the stack. This ensures the fast response time cycle needed for real-time reactive behaviors. Each active behavior is placed on a stack, the control loop cycles through the stack checking if the behavior should fire. The first behavior that fires (and only that behavior) gets control of the robots movement for that cycle.

In order to study the impact of the ERNI interface on teams of robots, we developed simple behaviors to control the robot movements: turn, randomwalk, goto, forward, backward and avoid obstacles. These behaviors are

designed to execute autonomously on a robot. The robot's RobotMotionResource inserts a behavior on the stack upon receipt of a BotMotion task with an appropriate sub-task (preposition).

4.2.3. Network aware behavior. All BotMotion tasks on a robot originate from its MissionPlugin, making it the seat of autonomous motion control logic. Mixed initiative behavior results when locally generated BotMotion tasks complement tasks received from remote agents. In our testbed we implement network aware behaviors using mixed initiative control. Each robot's MissionPlugin monitors the network for changes and reacts accordingly.

For example, when executing a high-level task from a remote controller, the robot may want to ensure network connectivity. When the robot becomes an articulation point of the network or when the quality of the best link to a neighbor goes below a threshold, it issues a BotMotion:stop task. The resulting effect is that the robots carry out the user-directed high-level task while maintaining local network connectivity. Future work will measure the impact of mixed initiative control on task completion times and rates for distributed team missions such as distributed sensing and mapping.

4.3. Human Interactions

Human operators are supported via a user interface. The user interface is just another Cougaar plugin; it is the same as that on the robotic node except that it has no robot motion control plugin or resource. Because of the pluggable nature of our architecture a robot could easily be enhanced to have a UI.

There are two components that support human interactions with our testbed. Our traditional GUI supports both information display and robot tasking via buttons, popup menus and click-and-drag interactions. Our speech user interface supports hands-free, spoken language tasking of multiple robots.

4.3.1. The graphical user interface (GUI). Our graphical user interface to the robot system has two primary goals: remote control of multiple robots and visualization of the state of the robots as well as the wireless network topology and metrics. Figure 7 shows a screen capture of our interface. The numbered circles represent the robots' position in the environment; the extruding line shows the heading of the robot. The color of the robot indicates its current state: green means normal/active, pink abnormal/low battery. The user can drag a robot to a location on the grid to teleport the robot to a specific location. Robots can be tasked either individually by right-clicking on the icon or in groups by selecting them with the checkboxes and then using the command buttons. The window support panning and zooming.

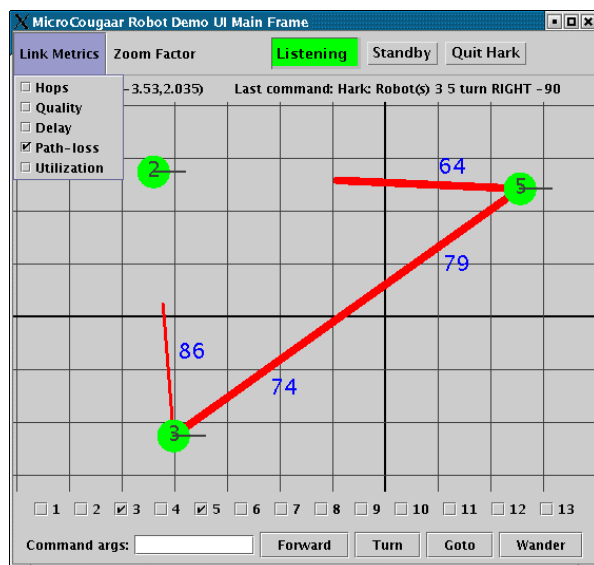


Figure 7. GUI used to remotely control/monitor robots from laptop or iPaQ.

4.3.2. Implementing a speech-based interface for robot control. Controlling teams of mobile robots is a complex task for a human operator. As robots become more autonomous the burden of teleoperation is greatly reduced, freeing the operator to engage in other cognitive activities such as mission planning, collaboration, etc.

A spoken language interface is well suited to autonomous robot tasking. Speech provides a natural mode of communication among collaborators. Speech offers the user a hands free, eyes free interface that is particularly well suited to mobile environments. Tasking semi-autonomous robots in an on-the-go environment such as might be found in military situations is particularly well suited to spoken language interface.

We implement a simple speech interface that complements the GUI by allowing the user to task specific robots using voice and/or the GUI. Figure 8 shows the block diagram for our multi-modal user interface (MMUI). It consists of a single logical UI plugin. Separate sub-components manage input and output via the traditional GUI (screen, keyboard, mouse or stylus) and the speech UI (audio input and output).

The central UI plugin manages the communication with the rest of the ERNI system via publications and subscriptions to the Cougaar blackboard. It also keeps the GUI updated with the latest status information for the network and the robots.

The speech UI sends audio data read from the local microphone to the speech recognizer. Currently, we use a TCP socket to send audio data to the server which can be either local or remote. The iPaQ ships that audio data over the wireless ad-hoc network to a speech recognizer running on a floating point enabled laptop machine. In the

future we plan to use a real-time streaming transport, i.e., RTP, which works better over wireless links.

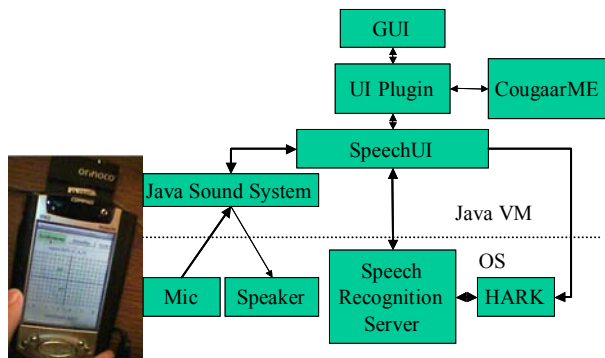


Figure 8. UI on an iPaQ and block diagram of the multi-modal user interface.

In summary our multimodal user interface is well suited to autonomous robot tasking because: 1. it offers multiple levels of control: direct manipulation and task level delegation; 2. speech commands map nicely into our task language; 3. as implemented on the iPaQ it provide a mobile lightweight user interface.

The speech recognition grammar available for controlling the robots is very simple. The format of all speech commands is: [*<subject>*] *<verb>* *<object>*. The user can also tell the speech interface when to listen and when to ignore their voice. This is critical in environments where extraneous speech can be misinterpreted by the speech recognizer.

Below is a typical interaction sequence with the speech interface:

Operator: "Attention!"

System: "Listening"

Operator: "Robots 5 and 6 turn right 90 degrees".

System: "Roger"

Operator: "Go forward 2 feet."

System: "Roger"

Operator: "Go to sleep"

System: "Standing by."

Once the user has specified the target robot(s), they do not have to explicitly name them again. The UI will assume that subsequent commands refer to the last targets until you explicitly specify new ones.

4.3.3. Audio-visual feedback. Robots don't have very self-revealing state; to make them more expressive in a hands-busy, eyes-busy environment we add audio feedback. We use audio cues to indicate internal state changes. Changes in network connectivity and reception of a remote-task evoke the playing of pre-recorded audio clips by the AmigoBot.

On the GUI we use persistent visual cues to indicate robot state. The color of a robot indicates status.

As shown in the example above, the speech UI also gives audio feedback on comprehension of input and to indicate the status of speech UI, e.g., listening or standing-by.

5. Limitations of architecture and future work

We believe that CougaarME is a good platform for embedded distributed applications. With some more enhancements to the transport layer services, for example, reliable UDP, CougaarME offers a lightweight, portable platform on which to develop distributed collaborative systems. Some of the major limitations of CougaarME are not limitations of Cougaar in general but are due to its lightweight nature. Since we ran on a J2SE VM we could take advantage of Java features that CougaarME does not. For example, we found that the use of floating point arithmetic as well as the `java.util` package to be quite useful.

Future work will look at the enhancements to the UI including indications of communications failure. We also would like to measure the performance benefits of network aware behaviors on standard tasks for robot teams, such as target re-acquisition.

CougaarME's use of longs for all task allocation results instead of generic objects forced us to make some awkward design decisions. We used parallel publications and subscriptions for intra-robot data sharing, e.g., `NetInfo` objects in addition to `NetWatch` tasks. For inter-robot communications we were constrained to using longs for reporting all values. This limitation could be removed by adding support for generic object serialization as is done in full Cougaar.

Finally, because we could not depend on a centralized name server to hold the `MicroAgent` directory, we were forced to configure each robot to know about all other robots. In a survivable ad-hoc wireless network, a central server is not a workable solution, however, pre-configuration is not scalable or flexible. Instead what is needed is a JXTA like agent discovery service that could leverage knowledge of the underlying network topology to maintain a distributed who's who list.

Some of Cougaar's features can also be liabilities. For example, it's pattern of use makes it easy to develop once you understand plugin design and development but presents a formidable learning curve to the first-time user [R5]. Similarly, Cougaar's proprietary messaging service makes cougaar to cougaar interactions transparent but prevents easy integration with foreign components. An OMG trading service model or equivalent might be a nice complement to the standard naming service. This CORBA-bridge, would enable Cougaar applications to benefit from the wealth of non-Java, non-Cougaar developers and vice-versa.

6. Conclusions

We have described an application of CougaarME for distributed control and monitoring of a team of semi-autonomous robots across a self-forming, self-healing multi-hop wireless network. In the process, we leveraged Cougaar's services for task distribution and reporting. Cougaar's publish and subscribe blackboard service was a great match for our enhanced robot mission - network interface (ERNI) since it enables designers of robot control algorithms to easily use information available to the ad-hoc network without being tied to a specific interface. We have also made some enhancements to the transport service to account for the unreliability of the network. Finally, we added a multi-modal interface enabling tasking of the robot team with spoken commands.

7. Acknowledgements

The authors would like to acknowledge Keith Manning, Manchui Lee, Ben Rubin and Marty Grossman for their help in developing the robot testbed. This work was sponsored by DARPA/IPTO under contract number DASG60-02-C-0060. Content of this work does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

8. References

- [1] J. Redi & J. Bers, "Exploiting the Interactions Between Robotic Autonomy and Networks," in *Multi-Robot Systems: From Swarms to Intelligent Automata*, Vol. II, A.C. Schultz *et al.* (Eds.), Kluwer Academic Publishers, pp. 279-289, 2003.
- [2] Prithwish Basu and Jason Redi, "Movement control for achieving fault tolerance in ad hoc robot networks," accepted for publication to *IEEE Network, Special issue on Ad Hoc Networking: Data Communications & Topology Control*, Eds: David Simplot, Ivan Stojmenovic, July/August 2004.
- [3] Evan Woo, Bruce A. MacDonald, and Félix Trépanier. Distributed mobile robot application infrastructure. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1475-80, Las Vegas, October 2003.
- [4] R. T. Vaughan, B. P. Gerkey, and A. Howard. "On device abstractions for portable, reusable robot code." In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 2121-2427, Las Vegas, Nevada, October 2003.
- [5] E. Gat. "On three-layer architectures." In D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots*. MIT/AAAI Press, 1997.
- [6] Volpe, R., et al., "The CLARAty architecture for robotic autonomy," *2001 IEEE Aerospace Conference Proceedings*, vol, 1, pp. 121-32.
- [7] D. Brugali and ME Fayad, "Distributed computing in robotics and automation," *IEEE Trans. Robot. Automat.*, vol. 18, no. 4, pp. 409-420, August 2002.
- [8] http://cougaar.org/docman/view.php/9/4/CougaarME_ArchV1.1.pdf
- [9] <http://www-robotics.usc.edu/~maxim/JavaClient/jc.htm>
- [10] <http://www.amigobot.com/>
- [11] <http://www.blackdown.org/java-linux/java-linux-d1.html>
- [12] <http://cougaar.org/projects/cougaar/>
- [13] <http://java.sun.com/j2me/index.jsp>
- [14] Zhenghua Fu; Xiaoqiao Meng; Songwu Lu, "How bad TCP can perform in mobile ad hoc networks," in *Computers and Communications*, 2002. Proceedings. ISCC 2002. Seventh International Symposium on , 1-4 July 2002 pp.298 - 303