ELSEVIER

Microprocessors and Microsystems 23 (2000) 597–607

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.nl/locate/micpro

The application of wireless local area network technology to the control of mobile robots

A.F.T. Winfield*, O.E. Holland

Intelligent Autonomous Systems (Engineering) Laboratory, Faculty of Engineering, University of the West of England, Bristol, Frenchay, Bristol BS16 1QY, UK

Received 16 September 1999; received in revised form 30 November 1999; accepted 15 December 1999

Abstract

This paper describes a communications and control infrastructure for distributed mobile robotics, which makes use of wireless local area network (WLAN) technology and Internet Protocols (IPs). The use of commercial off-the-shelf (COTS) hardware and software components, and protocols, results in a powerful platform for conducting experiments into collective or co-operative robotics. Standard Transmission Control Protocol/Internet Protocol (TCP/IP) compatible applications programming interfaces (APIs) allow for rapid and straightforward development of applications software. Further, the message bandwidth available from WLAN interfaces (1–2 Mbits/s) facilitates multi-robot experiments requiring high data rates, for instance in robot vision or navigation. The infrastructure described is equally applicable to tele-operated mobile robots. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Mobile robotics; Telerobotics; Wireless local area networks; Internet protocols

1. Introduction

Since the advent of high-performance wireless local area network (WLAN) technology at relatively low cost, its use for wireless control of mobile robots has become a practical proposition. When a number of mobile robots are required to co-operate then, from a systems design perspective, the possibility that each mobile robot might be treated as a node on a LAN is particularly attractive. Proprietary wireless LAN devices that make use of spread-spectrum modulation and a UHF carrier (typically 2.4 GHz) offer the potential for high message data rates over a reliable physical layer implementation. Further, the fact that such devices typically offer support for standard transport and network layer protocols, such as Transmission Control Protocol/Internet Protocol (TCP/IP), gives rise to the possibility of powerful multi-robot networks at a relatively modest cost in application level design and development effort.

This paper describes such an implementation on a fleet of miniature-wheeled mobile robots, designed with the overall goal of conducting laboratory experiments in collective or multi-agent robot behaviour and control [1]. It is not the intention of this paper to describe the collective robotics algorithms or experiments. Instead this paper focuses on the hardware and software architecture of both the mobile robot and the fixed control base station with particular reference to the wireless control and communications infrastructure, and its software.

This paper proceeds as follows. First we review the development of wireless local area technology. The paper then describes the hardware architecture adopted for the mobile robot and in particular the choice of an IBM PC compatible embedded micro-controller. Following the discussion of the choice of operating system, the paper then describes the software architecture with particular reference to the use of TCP/IP communications protocols and the choice of a client–server paradigm for multiple robot command and control. Finally the paper comments on the performance achieved in the laboratory by the WLAN-based robot controller.

2. Wireless local area network technology

WLAN technology, developed primarily to extend wired networks to allow, for instance, roaming network nodes for portable computers within a building, is now relatively mature [2]. Typically a WLAN connection will employ spread-spectrum modulation over a 2.4 GHz RF carrier, with a raw over-air data rate of 1–2 Mbits/s. Spread-spectrum

* Corresponding author.

E-mail addresses: alan.winfield@uwe.ac.uk (A.F.T. Winfield), owen@micro.caltech.edu (O.E. Holland).

PII: S0141-9331(99)00074-5

modulation is a technique that, as the name implies, disperses the modulated signal over a much wider RF bandwidth than the conventional modulation techniques. Spread-spectrum modulation is particularly appropriate for a conventional WLAN environment because it helps overcome problems that would normally be associated with multiple transceivers sharing the same RF spectrum, and with high levels of multi-path interference. Spread-spectrum modulation also confers a high degree of noise immunity, including immunity to accidental or deliberate interference.

There are two variants of spread-spectrum modulation in common use. Frequency hopping (FH) spreads the spectrum by rapidly switching the carrier frequency. The more sophisticated direct sequence (DS) technique achieves the same effect by multiplying the message data with a pseudo-random bit sequence (PRBS) [3]. Both variants have the same overall characteristics outlined here, but DS typically will allow a higher over-air data rate than FH.

Because of their intended application with portable or notebook computers, manufacturers have produced remarkably compact wireless network interface hardware. Typically these employ the Personal Computer Memory Card International Association (PCMCIA) interface [4], which is a de facto standard in portable computers[1], and usually have a two-part construction consisting of a PCMCIA card with a separate similarly sized wireless transceiver. Two additional implementations are available: an ISA-bus plug-in card for desktop computers, and a stand-alone wireless to wired network bridge, sometimes known as an Access Point. It is the PCMCIA wireless network interface that is of particular interest here, since its compact size makes it ideal for integration into an embedded micro-controller suitable for mobile robotics applications.

Although wireless devices for interconnecting computers and their peripherals have been available for some years, it is only recently that the adoption of agreed standards by manufacturers has meant that the wireless LAN can be regarded as a generic system level component interchangeable with the wired LAN network interface card (NIC). Two standards in particular have brought this about: the IEEE standard 802.11-1997 [5] and the European Telecommunications Standards Institute (ETSI) spectrum allocation RES.2 [6].

The IEEE 802.11 standard specifies the media access control (MAC) protocol, which forms the lower half of layer 2 of the open systems interconnect (OSI) 7-layer network reference model, and also the physical layer (PHY) specification for layer 1. Within the physical layer specification of IEEE 802.11 there are standards covering the use of either infrared optical communications or spread-spectrum radio. The spread-spectrum radio standard in turn covers both FH and DS variants.

[1] Now often referred to as simply the PC-card interface.

3. Hardware design considerations

A fundamental design decision was to employ a proprietary microprocessor controller based upon the IEEE PC/104 standard [7,8]. With PC/104 we have a form factor small enough to be accommodated within the overall physical requirements of a small mobile robot: a standard PC/104 card measures 90 mm × 95 mm. More significantly PC/104 provides us with a micro-controller based upon the Intel 386, 486 or Pentium processor within an *IBM Personal Computer* compatible architecture. Clearly a card measuring about 14 in.^{2} cannot accommodate a complete PC and so the PC/104 card set provides a modular architecture in which major functional blocks are all implemented as different PC/104 cards, all interconnected via the PC/104 bus. This modular approach clearly suits our mobile robotics application, since many sub-systems of the standard desktop PC architecture are unnecessary—the VGA (display) controller for instance. In fact a CPU module typically embodies the 386 or 486 microprocessor, together with its supporting logic: RAM (typically 4 MB); the basic input output system (BIOS); serial and parallel I/O; floppy and IDE (hard disk) interfaces; and a socket for a flash EPROM or solid-state disk device. This is sufficient for a complete single-card embedded micro-controller.

The flash ROM device merits further discussion since it is here that our mobile robot control and communications application software will be held. To understand how the application ROM is utilised we need to consider that in a standard PC, after power-up, the BIOS looks for a disk-drive from which to boot load the disk operating system: on a conventional PC this is either a floppy- or more usually a hard-disk drive. Assuming that we are running MS-DOS, then the boot up process requires that the executable files associated with MS-DOS (command.com, msdos.sys and io.sys) are all present, together with the script files autoexec.bat and config.sys, and any additional drivers required by the hardware or by the application. By arranging for all of the system files expected by the operating system to be present within the device together with the application code executable (itself started from within autoexec.bat), then we can treat the ROM as a ‘virtual’ (i.e. solid-state) disk drive.

The recent availability of high capacity *IDE compatible* solid-state disk drives has opened up the possibility that we can use operating systems other than MS-DOS within the mobile robot, for example Linux. An 80 MB IDE solid-state disk drive typically measures 75 mm × 50 mm × 10 mm, weighs 45 g and consumes about 75 mA at 5 V. The current generation of 386 or 486 based PC/104 processor card typically incorporates an IDE interface and so we can realistically mount a solid-state disk drive within the mobile robot with only a marginal increase in mass and energy consumption. An additional major advantage of solid-state disk technology is its ability to withstand much more severe physical shock than conventional disk drives. Collisions are a normal operational hazard of experimental mobile robots,

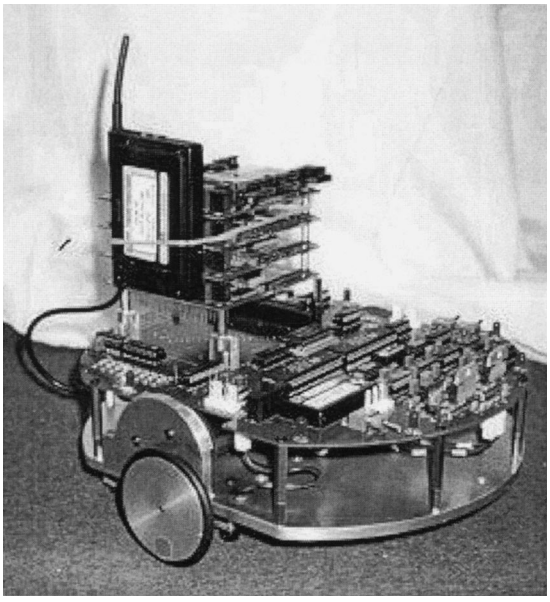


Fig. 1. Mobile robot with PC/104 stack and WLAN.

and conventional disk drives would require mass-increasing ruggedisation to ensure reliable operation.

In its minimum configuration the mobile robot requires one additional PC/104 card: a PCMCIA adapter, which will allow us to make use of a PCMCIA wireless LAN Network Interface Card.

The PC/104 bus does not specify a physical backplane, but instead employs a plug and socket arrangement which allows PC/104 compliant cards to be 'stacked' one on top of the other. Hence the provision of a PC/104 bus interface and connector on the robot 'motherboard' allows for the CPU and PCMCIA cards to be neatly stacked above the motherboard as shown in Fig. 1. (The solid-state disk drive is small enough to be mounted *between* the motherboard and the underside of the PC/104 stack.) The combination of the physical connectors and spacing pillars provides a robust construction capable of reliable operation while subject to the physical stresses (in particular accelerations, decelerations and collisions) inherent in a light wheeled mobile robot.

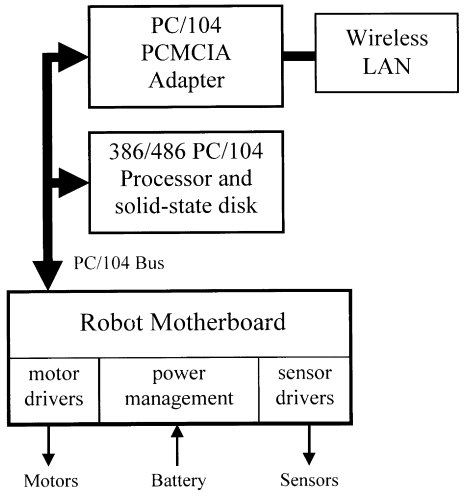


Fig. 2. Hardware block diagram.

Fig. 2 illustrates in block diagram form the bus architecture described above. The motherboard shown in Fig. 2 provides all of the motor drive power electronics, power management (regulation and battery level monitoring) and analogue and digital interfaces for sensor devices that may be mounted on the mobile robot. The PC/104 compliant interface on the motherboard maps the input and output interfaces to the robot's motors and sensors into the CPU's I/O space; all robot functions can then be controlled using straightforward I/O port reads and writes using, for example, the C functions *inportb( )* and *outportb( )* (or, under Linux, *inb( )* and *outb( )*).

The use of a PC/104 compliant micro-controller confers a remarkable power and simplicity to the robot control architecture. Not only does it allow us to make use of the powerful off-the-shelf PCMCIA compliant WLAN technology, but also most significantly, it provides an embedded controller that, from a software development perspective, is *identical to a desktop PC*. Not only can we then use standard PC development tools, including for example familiar C compilation and debugging environments, but most importantly, we will be able to use off-the-shelf MS-DOS or Linux compatible hardware device drivers for the PCMCIA and WLAN hardware. Our software development effort is free therefore to focus only on the high-level robot control applications development without the burden of low-level device programming. Software development for the mobile robot is, as a consequence, rapid and straightforward.

4. Operating system considerations

As described above, the PC compatible embedded microprocessor controller chosen for the mobile robot gives rise to a number of options in the choice of robot operating system, and we have successfully made use of both MS-DOS and Linux. While MS-DOS may be uncontroversial for a minimal embedded system, the use of Linux is less obvious, and so merits further discussion.

Linux is a Unix-like multi-user multi-tasking operating system that is available for a very wide range of hardware platforms, including the PC [9]. Linux has a number of attributes that make it particularly attractive for the application described in this paper. In particular:

- Multi-tasking is particularly useful in a robot controller so that, for instance, a number of robot 'behaviours' can be implemented as separate tasks or processes, any of which may be started or stopped independently of the others.
- Linux integrates TCP/IP networking as standard, together with session or presentation layer protocols

such as the file transfer protocol (FTP), Telnet, or the hypertext transfer protocol (HTTP). Thus Telnet facilitates remote control or monitoring of any individual mobile robot, by allowing a user to 'log-in' to the robot via the wireless LAN. By the same token FTP assists program development or results collection by allowing data to be transferred to or from a mobile robot via the WLAN.

- Because of the open source code policy of the Linux developers the internal operation of Linux is transparent, unlike proprietary operating systems. This is especially useful when developing new low-level hardware device drivers, such as for the WLAN card [10]. In fact the open source nature of Linux means that there are a very large number of device drivers in the public domain, including drivers for many PCMCIA WLAN cards.
- Although Linux is not intrinsically a real-time operating system, a number of real-time extensions to the Linux kernel exist which allow time-critical tasks, or interrupts, to be precisely scheduled [11].

Equally compelling is the fact that Linux does not require large memory resources or a high-performance CPU (by current standards). In fact, we have successfully ported a full Linux implementation including all networking components, a C compiler and its libraries, onto the mobile robot hardware described above, comprising an Intel 386SX CPU card running at 25 MHz, with 4 MB RAM and an 80 MB solid-state disk drive. Linux also lends itself to optimisation. The Linux kernel may, for instance, be re-compiled to omit the drivers or services not required in a particular implementation [12].

5. Internet protocols

Consider the OSI 7-layer network reference model shown in Fig. 3. This provides a powerful model for describing discrete network 'layers' which allow us to 'mix-and-match' different network software components. The interchangeability of network software components is achieved by the adoption of standard interfaces between each layer. An implementation of the network layers (3 and 4) is sometimes referred to as a 'protocol stack', which needs to be present at both ends of the communications link; in our case the mobile robot and the desktop controller. Any message from the applications layer at one end of the link (an instruction from the control PC for the robot to move, for instance), is transferred down the protocol stack at the originating end of the link, then across the physical network interface (in our case the wireless connection), and finally up the protocol stack at the destination (i.e. the robot). The arrows in Fig. 3 illustrate this message flow.

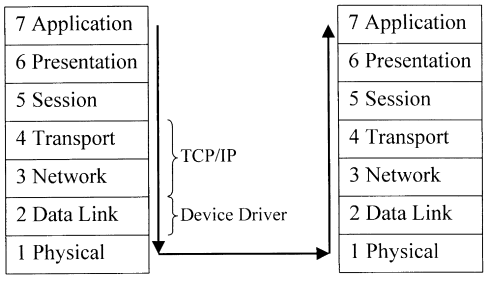


Fig. 3. OSI 7-layer network reference model (showing both ends of a switched virtual circuit).

Layer 2, the data link layer, is represented in software by the 'device driver' which a manufacturer needs to supply with the network interface hardware. Layers 3 and 4 are frequently grouped together and given a generic network description. The group of protocols known as the Internet Protocols, or TCP/IP, have arguably become the most widely used for both local and wide area networks [13]. Clearly, one of the most interesting benefits that might flow from the adoption of TCP/IP is that the mobile robots could send and receive messages to and from anywhere with Internet connectivity. This would allow the full spectrum of operational possibilities, ranging from direct tele-operation, through assignment and monitoring of missions, to passively receiving data from fully autonomous robots.

Even if remote operation via the Internet is not a requirement, there are still strong technical arguments in favour of the use of TCP/IP. One is the fact that the protocols are well known and understood, with well-established libraries to support the applications programmer. Another is that standard and proven software components to implement TCP/IP are available for practically every operating system in common use. We have, for instance, employed both MS-DOS in the mobile robot controller, with TCP/IP software components from FTP Inc, and more recently Linux, which has the TCP/IP networking built-in. The controlling computer may typically employ MS Windows 95/98 or NT, both of which have built-in support for TCP/IP. The fact that different operating systems can be employed at each end of the link might also be regarded as an advantage.

A particularly strong argument in favour of TCP/IP is that the TCP employed in the transport layer provides us with a robust and reliable data connection. In contrast to the alternative User Datagram Protocol (UDP), error detection and repeat request mechanisms are built into TCP so that, providing the connection is not physically broken, reliable data delivery is practically guaranteed. This means that the applications programmer does not need to be concerned with data integrity. Once a TCP connection has been established data can be transmitted without the need for acknowledgements or other such handshake mechanisms in the applications layer code. In short the development engineer does not need to 'invent' a reliable communications protocol, as would be the case for a completely bespoke radio telemetry link.

6. System software design

Equipping a fleet of mobile robots with WLAN interfaces offers two distinct opportunities: enabling communication between robots; and enabling communication between robots and a base station. The focus of our laboratory is on collective and co-operative robotics, and so it might be thought that communication between robots would be the first priority. However, this is not the case; the paradigm in which we are most interested is that of swarm robotics, in which communication between robots is minimal, usually taking the form of low-bandwidth broadcasting using infrared or ultrasonic channels [14] or direct sensing of the environmental disturbances created by other robots [15]. Instead, our first application of this technology focussed on the communication between robots and a base station, or controller.

The overall system described in this paper employs a client–server paradigm. Although it may seem counter-intuitive, the desktop base station at the hub of the WLAN acts as the 'client' software, and the application software in each mobile robot acts as a 'server'. The reason for this approach is that in the control architecture implemented here each robot will passively 'listen' for commands from the base station. When a command is received (to start or stop a motor drive, for instance) the robot will execute the command immediately, then (if the command requires it) the robot will respond with a reply message. Finally the robot will simply return to its passive listening state (of course the robot may actually be on the move while in this listening state). Thus the robots passively service commands sent by the controller and are, from a networking perspective, 'servers'. The 'client' software at the network hub issues commands to each robot, either from manual control, or from some 'script', and is thus entirely responsible for the sequencing and timing of commands, and the coordination of the actions of different robots.

Of course the alternative control schema could be envisaged in which robot 'clients' actively fetch or request their next commands from a central 'server'. However, such a scheme was considered inappropriate for this first implementation, especially since manual control (for test and debug) would be more difficult. Another advantage of the robot–server, hub-client architecture adopted here is that, within the same control paradigm, different levels of robot autonomy could readily be implemented. In other words, commands sent from the network controller could be at the low level of motor-on or motor-off (as in this implementation) but could easily be at higher levels of control abstraction such as *go and execute this given trajectory* or, at a still higher level, *navigate to point C via way points A and B.*

Fig. 4 illustrates the network architecture showing the central controller (client) and a number of mobile robots (servers). As illustrated, each robot must have its own unique IP address.

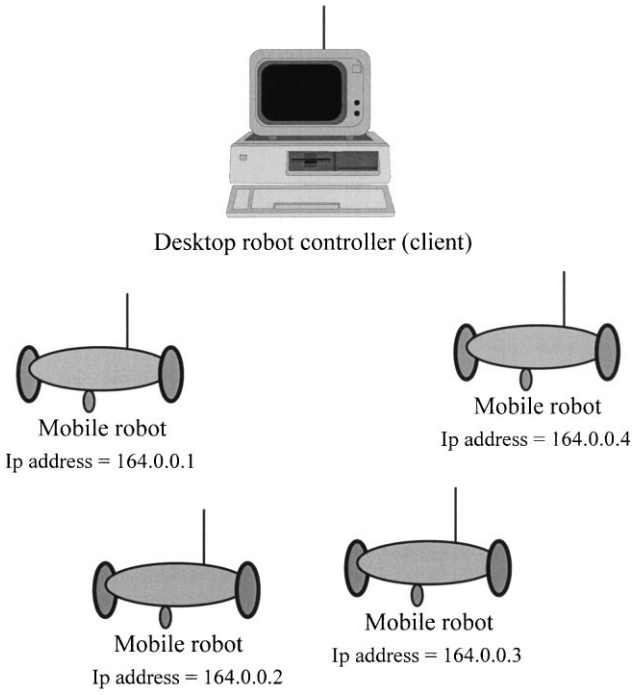


Fig. 4. Multi-robot wireless local area network.

6.1. The applications layer command protocol

The use of the reliable TCP communications protocol means that, at the applications layer, there is no need to incorporate either error detection or retransmission protocols. In fact TCP is sufficiently robust that a command can be sent to a robot (to start a motor with a given speed, for instance), without the need for the robot to respond with an acknowledgement message. Indeed, for the same reason, there is no need for message length or checksum data to be incorporated into the command string. We can therefore construct a very simple command protocol in which some commands consist of a single message from the controller to the robot (motor commands, for instance). Other commands consist of a request message (to read a proximity sensor, for instance) followed by a response message from the robot back to the controller (i.e. the sensor value).

Both for ease of debugging, and to simplify command parsing in the Robot Server software, each command from controller to robot is encoded as a fixed length ASCII numeric string. Some commands require only the numeric string, but others require parameters (motor speed values, for instance), and these are also encoded as fixed length ASCII strings, separated by spaces. The Robot Server software will be able to determine from the command value whether there will be parameter values and how many. Table 1 lists a few such commands, with their optional parameters and responses.

Certain commands require further explanation. The NULL command, for instance, consists of the two ASCII characters '00', and is intended to solicit an identical '00'

Table 1
Controller to robot command values

Command	Value	Parameters	Response	Comment
NULL	00	None	Yes	Keep alive
STATUS	04	None	Yes	Read status switches
MOTOR	06	Two (left, right)	No	Set motor speeds
LEDS	07	One (data)	No	Set LED values

response back from the robot. This command serves two purposes: firstly it allows the client application to periodically check that a robot is still responsive (at the applications level), and secondly it will keep the TCP connection alive (which might otherwise time-out after a long period of communications inactivity). In this implementation the network controller application sends a NULL message to each robot once every 10 s. The client application also notes the time of transmission of each NULL message and, by noting the arrival time of the corresponding response message enables transmission 'round trip' times to be measured.

The MOTOR command consists of the two ASCII characters '06' followed by the 'left' and then the 'right motor' speed values (the mobile robot is a differential drive robot with independent motor drives on each wheel). The speed values are signed so that the command string '06 00100 00100' would mean 'forward at speed 100', the command string '06 00050 –0050' would cause the robot to rotate on the spot, and the command string '06 00000 00000' would be interpreted as 'all stop'.

Note that messages do not need to incorporate any form of identifier, or 'unit number', for the particular robot for whom the message is intended. This is because each robot has its own unique connection, or SVC, with the client network controller. The client application sends a command message to a given robot simply by sending the message packet to that robot's unique SVC. Only the single robot connected via the SVC will actually receive the intended message packet. Similarly, in the reverse direction, any response message from a robot server back to the client application will only be received via the unique SVC associated with the robot. The client therefore 'knows' which robot sent the response from the SVC on which it was received.

The command set is intended to be simple and extensible, allowing, for instance, new commands to be created as new sensors or actuators are added to the robot hardware. This paper is not, however, concerned with a detailed discussion of which commands and actions might be required in a given application scenario, and the representative commands listed in Table 1 are given only to illustrate the simplicity of the applications layer protocol.

6.2. The Robot Server software

As already described the mobile robot incorporates a canonical PC compatible microprocessor controller board running MS-DOS or Linux. Consider first MS-DOS. In addition to the proprietary device drivers required to support the PCMCIA card carrier and the wireless LAN card, an MS-DOS implementation of the TCP/IP 'protocol stack' is required, and the FTP Inc PCTCP product [16] was selected for this purpose. The FTP Inc software is a straightforward implementation of the 'Berkeley Sockets' [17] application programmers interface (API) between the application and the TCP (or UDP) layer. This product has the additional (and necessary) feature of providing the 'include' and library files for the major PC C language compilers.

For the Linux operating system we also require device drivers for both the PCMCIA card carrier and the WLAN card. However, no additional TCP/IP support is required since TCP/IP networking is an integral part of Linux. Linux also provides an API that conforms to the Berkeley Sockets model. Thus, the application code for the robot server is practically identical for both MS-DOS and Linux. The only significant difference is that the robot server, under Linux, runs as a background task in parallel with other network servers such as FTP and Telnet, whereas in the single-tasking MS-DOS the server is the only program running.

The application or Basic Robot Server (BRS) software consists of some 250 lines of C source code. The structure of this software is very straightforward, with three distinct phases. The first is an 'initialisation' phase in which the sockets and associated data structures are created. Secondly it enters a 'listening' phase, in which the server is waiting for a connection (SVC) to be established by the network controller (client). The third phase is a quasi-infinite loop that waits for commands from the client, then executes them when they are received.

The pseudo-code shown in listing one below (basic robot server pseudo code) illustrates the structure of the BRS application with particular reference to the calls to the Berkeley Sockets API functions, which are shown in bold text.

Listing 1

```
// PHASE 1, initialisation..
// Get a TCP stream socket, for listening..
socket (…);
// Get the local (our) IP address..
gethostid ( );
// bind our local IP address & port number to the socket..
bind (…);
// "passively open" the socket
listen (…);
// PHASE 2, listening..
// wait for Client controller to 'connect'..
accept (…);
// now close listening socket (not accepting any more
connections)..
```

```
close (…);
// set the connected socket to non-blocking..
ioctl (…);
// PHASE 3, Loop checking for received commands from
Client Controller..
while ( )
{
  // Execute a non-blocking read..
  if read (…)
  {
    // We've received a command, so go and act on it..
    // first parse the command value
    cmd = strtol (…);
    switch (cmd)
    {
      case NULL:  // echo null message
        send (…);
      case STATUS:  // read status switches
      case MOTORS:  // set motor speeds
    }
  }
  else  // nothing has been received, so perform local
  control here..
} // end of while loop
```

Listing 1

For clarity, the data-structures are omitted from the pseudo-code in the above listing; likewise, the error checking on return values from each of the sockets API calls is not shown. A number of aspects of the pseudo-code merit further explanation. Firstly, note that the initial call to **socket( )** returns a socket which is used to listen for the connection. (A 'socket' is simply a virtual input–output port for TCP packets.) The listening state (phase 2) is essentially in the call to the function **accept( )**. This is a blocking call, which means that the function does not return until the client has established the connection. When **accept( )** does return, it returns a new socket which is used thereafter as the SVC connection between the client and this server software. This explains why the original 'listening' socket is then closed. The new 'connected' socket is then set up as non-blocking by the call to **ioctl( )**, so that the subsequent calls to **read( )** will return immediately whether or not data has been received.

Within phase 3, the while loop, we have the non-blocking call to the **read( )** function. If a message has been received on the connected socket then **read( )** will return 'true', with the received data as an ASCII string. Note that in the pseudo-code above the standard C library function strtol( ) is used to 'parse' the received message string, to convert the leading numeric digits into the command value 'cmd'. A switch structure is then used to select the control code corresponding to the received command value. Alternatively, if nothing has been received then **read( )** will not wait, but return 'false', so that the BRS software can continue to execute local control or sensor monitoring functions in the 'else' clause of the 'if **read( )**' structure.

6.3. The Robot Client software

The Client software, running on the desktop PC at the hub of the WLAN, consists of some 550 lines of ANSI compliant C code. Of this code some 75% is responsible for managing the 32-bit Windows 95/NT graphical user interface (GUI). It is the other 25% which is of particular interest in this paper, since it is this code which is responsible for managing multiple TCP/IP connections with the mobile robot servers.

To implement TCP/IP compliant networking, the Client code makes use of the well-known Windows implementation of the Berkeley sockets API known as *Winsock*. The specification for the Winsock API is in the public domain [18] and its associated libraries and header files are distributed with most Windows C Programming environments, including Borland C++ as used for this work. While programming using Berkeley sockets is generally regarded as challenging, developing Winsock applications is even more so because of the need for asynchronous event handling [19]. To understand why this is so we need to consider, firstly, that all Windows applications are based upon an event-handling paradigm, and secondly that, because of the way Windows performs multi-tasking, all calls to I/O functions must be non-blocking. Any blocking call that did not return immediately would be likely to cause the whole of Windows to hang. Thus the Client software consists (as does any Windows program) of a series of event handlers each of which responds to messages posted by the operating system. Some of those event handlers (responding for instance to user input) will initiate non-blocking calls to Winsock functions, and other event handlers will in turn respond to the subsequent messages caused by asynchronous completion of these Winsock functions, or indeed by responses from the remote TCP connection.

It follows from the above that the Client needs to maintain a state-machine for each of the robots in its fleet. Each state-machine will reflect the status of its respective TCP connection and will move between the following states: *trying* (i.e. a TCP connection has been requested); *connected* (i.e. a SVC exists between client and server); *writable* (i.e. the server can accept messages from the client), and *disconnected*.

In order to appreciate the overall structure of the Client software a good starting point is the human machine interface (HMI) since, in this prototype system, the emphasis is on a user directly controlling robots via the base station. The next section therefore reviews the HMI for the client application.

6.4. The robot client human machine interface

There are four basic requirements for the HMI:

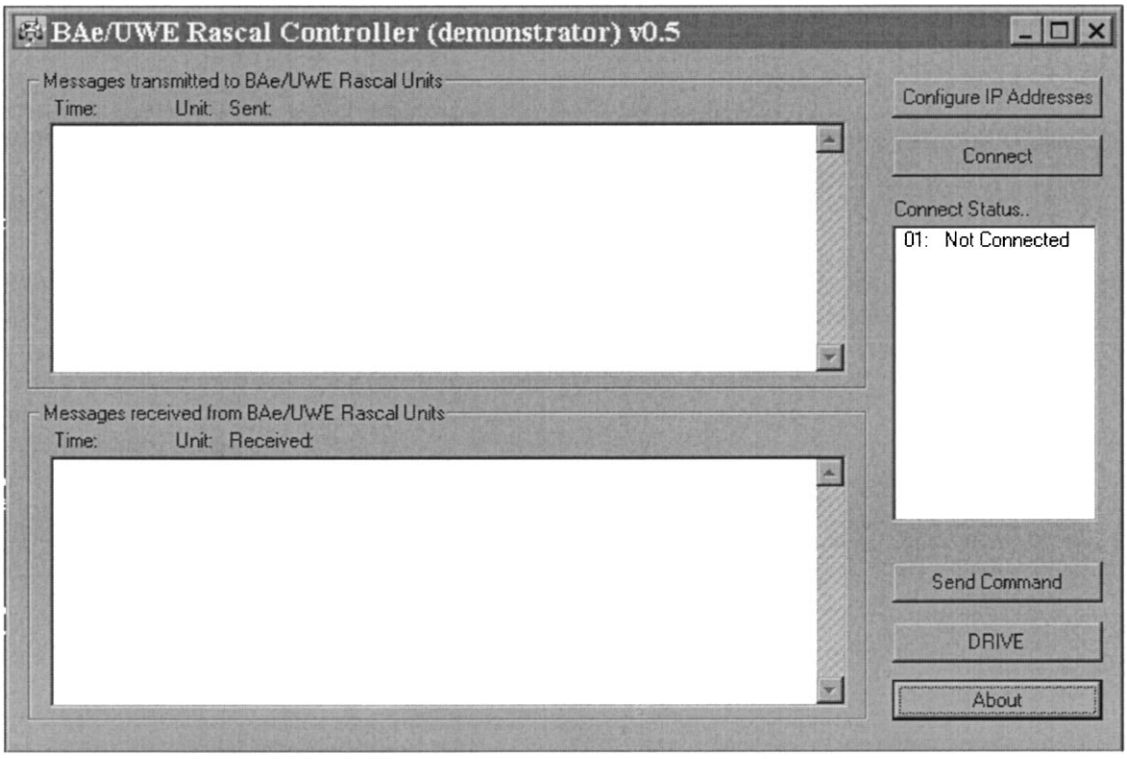


Fig. 5. HMI main window.

- the means to configure and manage IP addresses for the fleet of mobile robots;
- the means to initiate connections (virtual circuits) with each mobile robot, and to monitor the connection status of each virtual circuit;
- the means to send commands to any particular mobile robot; and
- the means to monitor both outgoing (command) and incoming (response) messages.

The Windows HMI meets these four requirements with a main (default) window, which is always open, and two subsidiary 'dialog boxes'. The main window provides the user with the ability to continuously monitor all messages to and from mobile robots, together with the connection status of each virtual circuit, as shown in Fig. 5. Clicking the 'Configure IP Addresses' button from the main window brings up the dialog box shown in Fig. 6. From this dialog box we can enter the IP addresses for each mobile robot, and associate each IP address with a 'unit number'. The unit number simply provides a shorthand means to identify each mobile robot.

Once the IP addresses have been configured, each mobile robot's unit number will be shown in the 'connect status' box of the main window as *disconnected*. When the 'Connect' button in the main window is clicked, the application will attempt to establish virtual circuits with each of the mobile robots in the IP address database. The 'connect status' box will initially show *trying* against each unit number, and these will in turn change to *connected* as each virtual circuit is successfully established. Of course each mobile robot must be in the 'listening' state, as described above, in order for the 'connect' process to complete successfully.

As soon as the application has established connections with one or more mobile robots, it will automatically start to send NULL messages to each connected

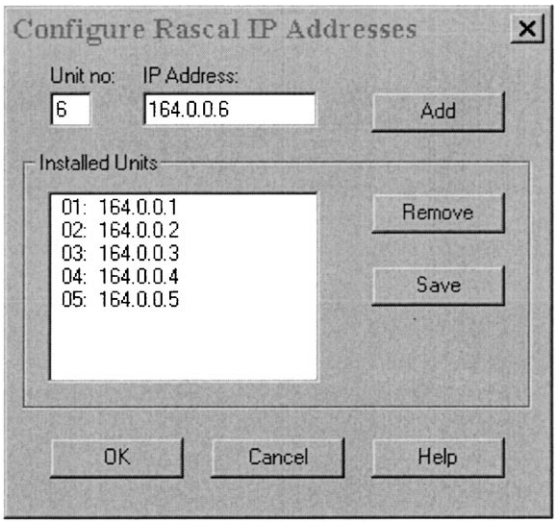


Fig. 6. Configure IP addresses dialog box.

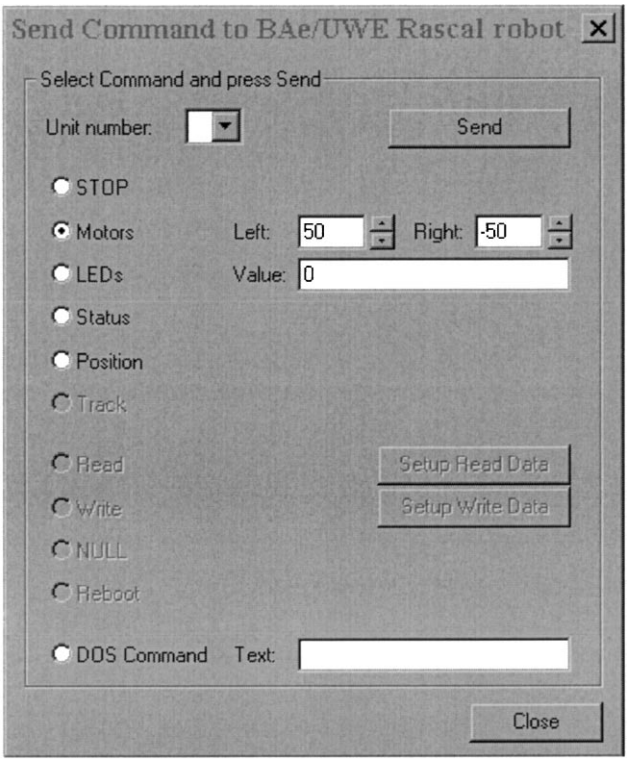


Fig. 7. HMI Send command dialog box.

robot, once every 10 s. Both outgoing and response messages are shown in the respective boxes of the main window, together with round trip times (in milliseconds).

While the overall robot network is in this state, the user may click the 'Send Command' button on the main window in order to bring up the dialog box shown in Fig. 7. From this dialog, the user is able to select one of the connected 'unit numbers', then select one of the available commands and finally click the 'Send' button to cause the command message to be transmitted to the mobile robot. The MOTOR command, for instance, allows left- and right-drive motor speeds to be entered via the controls in the dialog box. Altering the speed values and then clicking 'Send' a second time will transmit the changed values to the mobile robot. Selecting a command that provokes a response message, such as the STATUS command (which reads and returns a robot's status switches), will result in the response message being displayed in the main window's incoming message box.

Of course such low-level manual control of a fleet of mobile robots via this interface is not easy, particularly if a number of robots are moving simultaneously. This HMI is, however, designed for initial test and experimentation with the wireless network and, in particular, 'proof of principle'. A more sophisticated HMI would be required for serious experimentation with collective mobile robotics.

7. Results and conclusions

The key result from this work is that the proposed architecture has been demonstrated to reliably and successfully control a number of mobile robots, via the wireless network, in a laboratory environment. In fact, the system has proven to be sufficiently robust for demonstration at a number of conferences and science and technology exhibitions. The use of a notebook computer, fitted with an identical PCMCIA wireless LAN adapter to that installed into each mobile robot, acting as the network controller, has facilitated such demonstrations.

7.1. WLAN performance

A key performance indicator is the delay time introduced by the WLAN. In fact the measured average round trip time for a message to be sent from the control base station, to a mobile robot and then echoed back to the control base station, is 6 ms. A short message, consisting of a single IP 'packet', therefore requires of the order of 3 ms to be transferred across a SVC. This figure suggests that the control base station would, for instance, be able to command up to 10 mobile robots, communicating with each 10 times/s (allowing for a processing overhead in the control base station). A control loop frequency of 10 Hz is clearly too slow for low-level control functions in a mobile robot which is easily capable of moving at 1 m/s. However, it would not make sense to place the wireless communications within a low-level control loop, such as a PID or adaptive neuro-controller for individual motors. One of the benefits of having a powerful embedded microprocessor onboard each mobile robot is that many fast low-level control functions can be run entirely within the robot. Wireless communication is then reserved for higher-level control loops, supervisory or monitoring functions. In this way overall multi-robot control is *distributed* between the onboard robot control processors, the control base station and, if necessary additional high-performance processing resources connected to the control base station via wired-LAN.

7.2. Remote access via Telnet

Our laboratory is equipped with a WLAN Access Point, which provides a transparent bridge between wired and wireless networks. In practice this means that mobile robots equipped with WLAN interfaces and Linux (the *LinuxBots*) may be accessed from any of the fixed workstations on the laboratory LAN. This is especially convenient when a number of LinuxBots are operating in our mobile robot arena, since control software may be modified, compiled, executed and monitored entirely remotely, via the WLAN.

For direct access to any given robot in the arena we simply use *Telnet*. Once a Telnet connection is established with a particular robot (uniquely specified by its IP address), then we can log-in to the robot's onboard Linux operating

system. Providing the appropriate tools have been mounted into the onboard Linux, then we can edit, compile and run control programs in the usual way. In this fashion, programs to drive robot motors, read sensors and run simple behaviours such as obstacle avoidance or navigation can be developed and tested with minimal overheads. It is particularly convenient to be able to run such programs as background tasks, so that other tasks can be started and stopped in parallel. More sophisticated robot control experiments, especially those involving multiple co-operating robots, make use of the applications-level client–server architecture described in the main body of this paper. Of course it is still convenient to run simultaneous Telnet sessions with each robot, at the same time as the main client–server application, to facilitate low-level monitoring of system resources within each robot.

7.3. Simultaneous program execution, Telnet and Web access

The LinuxBot architecture additionally allows us to use the FTP to move program code or results data between robots and fixed workstations. This is particularly useful for 'backup' purposes, or to ensure that each robot in the collective experiment is running the same control software. Remarkably, we have also found that it is perfectly feasible to run the standard Linux *Apache* Web server as a background task within the LinuxBot. In practice a 25 MHz 386SX processor onboard a LinuxBot has been proven capable of running a number of robot control programs as background tasks (for navigation around an arena for instance), whilst simultaneously managing a number of remote Telnet log-in sessions and serving Web pages to a remote Web browser. The wireless communications and processing load imposed by Telnet and the Web server introduces no observable delay or interruption to the robot control behaviours running as background tasks. The ability to run a Web server within the mobile robot has opened up the interesting possibility that, by using Java, robot control and the HMI could be managed entirely through a remote Web-based interface.

7.4. Remote video tele-operation

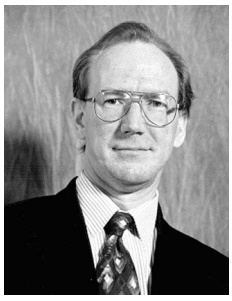
As well as facilitating the laboratory's work in collective and co-operative robotics, this technology has also enabled a recent strand of work concerned with tele-operation of a single robot equipped with a miniature digital camera. The introduction of a vision system places a significant demand on the bandwidth available from the WLAN system, but preliminary results do suggest that low frame-rate vision using wireless and TCP/IP is feasible within the architecture described in this paper [20].

Acknowledgements

The authors would like to gratefully acknowledge the British Aerospace Sowerby Research Centre, who supported the early part of the work described in this paper.

References

[1] R. Dawkins, O. Holland, A. Winfield, P. Greenway, A. Stephens, An interacting multi-robot system and smart environment for studying collective behaviours, Proceedings of the Eighth International Conference on Advanced Robotics, Monterey California, 1997.
[2] P.T. Davies, C.R. McGuffin, Wireless Local Area Networks, McGraw-Hill, New York, 1995.
[3] P. Wong, D. Britland, Mobile Data Communications Systems, Artech House Publishers, 1995.
[4] Personal Computer Memory Card International Association, PC Card Standard—Release 7.0, 1997.
[5] IEEE Std 802.11-1997, Wireless LAN Media Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE Publications, 1997.
[6] European Telecommunications Standards Institute, Spectrum Allocation RES.2, 1998.
[7] The PC/104 Consortium, PC/104 Specification version 2.3, June 1996.
[8] IEEE Draft Std P966.1, Standard for Compact Embedded-PC Modules, IEEE Publications, 1996.
[9] M. Welsh, K. Dalheimer, L. Kaufman, Running Linux, O'Reilly & Associates, 1999.
[10] A. Rubini, Linux Device Drivers, O'Reilly & Associates, 1998.
[11] M. Barabanov, V. Yodaiken, Real-time Linux, Linux Journal February (1997).
[12] M. Beck, H. Bohme, M. Oziadzka, Linux Kernel Internals, Addison-Wesley, Reading, MA, 1997.
[13] D. Comer, Internetworking with TCP/IP, vol. 1, Prentice-Hall, Englewoodcliffs, NJ, 1991 (and (with D.L. Steven) vol. 2, 1991 and vol. 3, 1993).
[14] C.R. Melhuish, O.E. Holland, S.E.J. Hoddell, Convoying: using chorusing to form travelling groups of minimal agents, Journal of Robotics and Autonomous Systems 28 (2/3) (1999) (special issue).
[15] O.E. Holland, C.R. Melhuish, Stigmergy, self-organisation and sorting in collective robotics, Journal of Adaptive Behaviour 5 (2) (1999).
[16] FTP Software Inc, PC/TCP Software Development Kit 3.2 for DOS, 1995.
[17] W.R. Stevens, UNIX Network Programming, Prentice-Hall, Englewood Cliffs, NJ, 1990.
[18] M. Hall, M. Towfiq, G. Arnold, D. Treadwell, H. Sanders, Windows Sockets: An Open Interface for Network Programming under Microsoft Windows, Version 1.1, January 1993.
[19] B. Quinn, D. Shute, Windows Sockets Network Programming, Addison-Wesley, Reading, MA, 1995.
[20] A.F.T. Winfield, Wireless video tele-operation using internet protocols, Proceedings of the Fourteenth International Conference on Unmanned Air Vehicle Systems, Bristol, April 1999.



Alan Winfield, shortly after completing a PhD in Digital Communications in 1984, gave up his lectureship at the University of Hull to found a company on the newly established Hull Science Park. Dr Winfield went on to establish APD Communications Ltd as one of the key UK providers of software for safety-critical mobile radio systems. He left APD in 1991 to take up appointment as Head of Research and Hewlett–Packard Professor of Electronic Engineering at UWE. Moving into the field of mobile robotics, he co-founded the Intelligent Autonomous Systems Laboratory in 1993. Current research is focussed on software and communications architectures for Distributed Mobile Robotics.

Owen Holland has a background in both psychology and engineering, and has held faculty positions in both disciplines. As well as having worked at universities in the UK, Germany, and the USA, he has substantial industrial experience in areas ranging from telecommunications to precision sensing. His interest in robotics dates from 1988, and he was a founder member of the Intelligent Autonomous Systems Engineering Lab at the University of the West of England in 1993. He is currently Visiting Associate in Electrical Engineering at the California Institute of Technology.