Asynchronous Network Control of Multiple Low Bandwidth Devices using Linux

Tully B Foote

March 18, 2006

Abstract

For driving an autonomous vehicle, network control of multiple low bandwidth devices is a necessity. For Team Caltech's entry into the DARPA Grand Challenge a program was developed for this purpose with the restriction of using open source software and consumer hardware. The result was an implementation written in C using the pthreads library. Near optimal control bandwidth was obtained. The program was successfully used to drive an autonomous vehicle at speeds up to 15 m/s.

1 Introduction

In the last few years, the use of autonomous vehicles has moved from an area dominated by pure research to a thriving applied industry. There are many situations in which an autonomous vehicle can outperform a manned vehicle or a remotely controlled vehicle. The advantage of an unmanned vehicle is that it does not have the limitation that a humans must be able to survive while on-board. For example an autonomous plane does not need to maintain pressure and provide oxygen for a pilot at high altitude. An intermediate step explored by the military was remote controlled unmanned vehicles. Remote control eliminates the need for protecting the pilot, however the remote control requires high bandwidth communication from the vehicle to its controller. The issue of bandwidth becomes very important in situations such as planetary exploration where communication with earth is severely limited by the distance. The other complication of requiring very high bandwidth communication is that while a few vehicles may be manageable, when the number of unmanned vehicles increases the bandwidth available becomes the greatest limitation.

The United States military is using more and more autonomous vehicles in an effort to prevent casualties and to cost-effectively extend its projected power. Since autonomous vehicles technology is still young, its capabilities are significantly lower performance than a human controlled vehicle. The Mars rovers are the work of a huge project that has been greatly successful at navigating autonomously on another planet. But the rovers travel at speeds measured in millimeters per second where as a human could drive on equivalent terrain on the order of 100 times faster. [8]

To stimulate growth in the field of autonomous ground vehicles, the Defense Advanced Research Projects Agency (DARPA), an arm of the United States Department of Defense charged with furthering research with potential military benefits, created a Grand Challenge. The initial concept was to hold an off-road race 200 miles long from Los Angeles to Las Vegas for autonomous vehicles. The team whose vehicle completed it the fastest in under 10 hours would win a cash prize of $1,000,000. The vehicles had to traverse the entire course where the only communication allowed was reception of publicly available broadcasts such as Global Positioning Systems (GPS) signals. The event was held in the spring of 2004 but no entry completed the race.[1] A second Grand Challenge was held in October 2005 with a cash prize of $2,000,000. This time multiple vehicles completed the race within the stated time of less than ten hours.[2]

At the California Institute of Technology, Caltech, generous corporate grants[4] and support from the engineering departments allowed the formation of a team of undergraduate students to enter a vehicle, named Alice, into the race. The team structure was built around classes and allowed students to work on the vehicle as an application of topics being taught. Working within limited budget constraints, Caltech's entrant into the second DARPA Grand Challenge was a completely new vehicle designed using knowledge from the first race. The chassis was a Ford E-350 that then was modified for off road use. For control, Alice contained a networked cluster of commodity Dell servers. And all the actuators either designed and built or bought were interfaced with RS-232 serial ports. On the servers Gentoo linux was chosen as the operating system. Distributed across the computers were software modules for each main function of the vehicle which communicated between computers using Spread [3] a messaging protocol. Dynamic path planning, trajectory following, stereo vision, and terrain mapping are a few examples of the functions of specific modules. All documentation of Team Caltech entrant can be found on their WIKI [5].

2 System Specifications

A module, called Adrive, was specified to provide an abstracted network interface between all the vehicle actuators and computer control. The primary role for the module was to listen for commands on the network then execute them within 50ms. The second role was to report regularly each actuator's current state (status and position). And a third role of the abstraction was to protect the vehicle from being damaged by control logic or system failures while testing, such as preventing acceleration and braking at the same time.

The computing platform available was Gentoo linux with standard development libraries running on 2.8GHz Pentium4 Dell servers. All of the actuators used to control the vehicle Alice used serial port interfaces. These interfaces ran at a range of speeds from the most common at 9600 baud up to 115k baud, with unique interface protocol for each actuator. To allow the vehicle to stop safely at speed the specification for the maximum allowable lag between receiving a command and execution was 50ms. The low bandwidth of the serial connections make this specification very challenging, however, there are several methods that were employed to help mitigate the limitations of the bandwidth.

The serial protocol was the limiting factor for the bandwidth to all the actuators. Consequently the design of Adrive was focused on maximizing the throughput of the serial communications. The first consideration was that commands need to go through as soon as possible to attempt to meet the 50ms delay specification. Secondly, regular feedback needs to be obtained from each actuator, but this cannot take up too much time using the serial port. Complicating matters was the fact that all of the actuators have a bidirectional flow of data for both commands and feedback.. Furthermore the transactions had to be completed before another one was started otherwise the communication would be assumed corrupted and no action would be taken.

3 System Design

To meet the specifications two less common techniques were combined. The first was that every major interface was run in parallel in an asynchronous interrupt driven manner. The second technique was to have two threads per interface. Using these two techniques allows a simple system to achieve near optimal speed of execution working with low bandwidth devices.

The standard Pthreads library for C based on POSIX,provides both the capability for simultaneous execution as well as an efficient method for setting and calling interrupts. [9] The simultaneous execution is not actually simultaneous but is time sharing the CPU on very short segments of time. Essentially each parallel operation is randomly chosen between. Although this could mean that a single function could negatively impact other pieces of code being executed on the machine, with good load management this was not a problem. [10]

To prevent any actuator blocking another actuator's execution all must be run in parallel. A simple solution would be to have one thread per actuator periodically forwarding commands and polling for status. This will provide a good response compared to communicating with each actuator sequentially, but using the second technique obtained higher bandwidth.

To obtain the higher bandwidth for each actuator, two threads were created for each actuator. The first thread periodically polled the actuator for its state and stored the latest position in memory. When position feedback was requested the latest positing was returned from the cached value. The actuator's second thread blocked on a pthread conditional until a command was received over the network. When the command was received, the thread immediately sent the command to the actuator. If the actuator was currently executing a command the new command could not send the command over the serial port because it was already in use; but it would record the command as the latest command. The ability to have the command thread waiting to immediately send data decreases the delay between receiving a command and executing it. Dropping unexecutable commands also increased bandwidth due to the the much higher bandwidth of the incoming commands. This is because before a previous command has finished executing over the serial port multiple new commands will have been received, and only the latest should be executed the next time.

Complications arose when making two threads communicate across a single port. To prevent collisions, communications to the actuator must be protected from conflicts between

the two threads. This was implemented using a pthread mutex. Whenever communications to the actuator was initiated by a thread it locked the mutex before it could use the port. However, if the mutex was already locked it will block until the mutex was released. Since there were only two threads running the only possible block was the complementary thread. Thus the maximum time that a command must wait was the remaining time in the request for feedback. Another consideration was that in the event that a command was being processed at the time the next command was received, the value will be recorded but not sent to the actuator. The risk is when no further commands are received. In this case the command thread has a timeout and will automatically execute the last command if a new command is not received before the timeout. The dropping of data is inherently required because of the higher bandwidth of the incoming commands than the actuator interfaces.

By only executing the latest command when the actuator is ready, instead of a method of queuing commands this will make subsequent commands have less lag. The maximum delay of the caching system is only $MaxDelay = 1/f_{actuator} + 1/f_{command}$ versus the minimum possible delay, $OptimalDelay = 1/f_{actuator}$, where $f_{actuator}$is the frequency of the actuator update and $f_{command}$ is the frequency of the incoming command. So as long as $f_{command}$ is much larger than $f_{actuator}$ this approach approximates optimal.

An optimal case can be seen by the example of three commands being received at twice the rate of possible execution. The first command will be executed, the 2nd command will be dropped since the 1st is still executing. But the 3rd will be immediately executed when received. Thus, as long as the frequency of command is significantly higher than the maximum frequency of execution this will be a good practice.

4 Implementation

The way that we implemented this system was through the use of a total of 19 threads. Our system had four actuators that required commands and gave feedback. To provide these functions all four had two threads. There were also two threads for two sensors that did not receive commands. Two threads were dedicated for communication over the network, one for receiving commands and one for broadcasting the latest state information. Two other threads were for the user interface and for a supervisory control. The rest of the threads were unnecessary but were useful as development aids such as for various logging methods as well as one thread that started all the other threads. The implementation can be seen in Figure 1.

All the current state data as well as configuration settings were contained in a hierarchical structure, called the vehicle struct. Within the vehicle struct each actuator had an associated struct, which contained the cached state data, configuration settings such as timeouts and which port, which was specific to the actuator. Any information relating to operation of the module was contained in the vehicle struct.

For each actuator a status thread and command thread were created. The status thread periodically polling the actuator for status while the command thread blocked until a new command was received and then executed immediately as described in the system design.

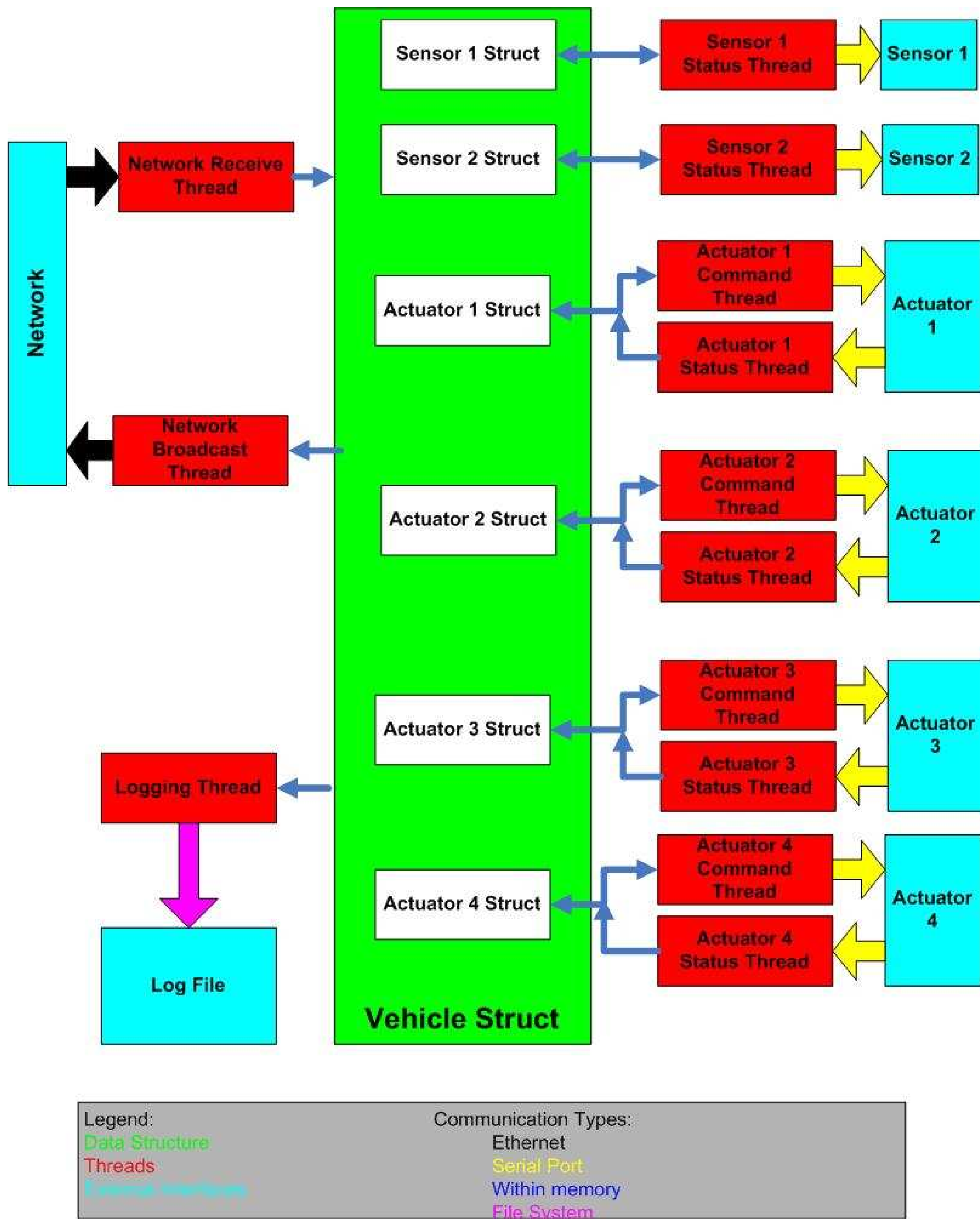

Figure 1: This is the data flow implemented in Adrive.

```
ADRIVE    EXIT     GUI_STATUS:       0
IF YOU HAVE NOT READ THE README IN THE LAST WEEK READ IT!  SHUTDOWN=     0
---------------------------------------+---------------------------------------
STEER -1(L) to 1(R)               0    | BRAKE  0 to 1                      0
 st delay: 250000  cm_delay:  100000   | st delay: 250000  cm delay:  100000
 status_e:      0  command_e:      0   | status_e:      0  command_e:      0
 status:        0  command:        0   | status:        0  command:        0
 position:      0  accel:          0   | position:      0  pressure:       0
 error:         0  velocity:       0   | error:         0
---------------------------------------+---------------------------------------
GAS      0 to 1                   0    | TRANS  0=park,-1=rev,1=drive       0
 st delay: 250000  cm delay:  100000   | st delay: 250000  cm delay:  100000
 status_e:      0  command_e:      0   | status_e:      0  command_e:      0
 status:        0  command:        0   | status:        0   error:         0
 position:      0                      | t_pos:         0    t_cmd:        0
                                       | i_pos:         0    i_cmd:        1
                                       | l1_pso:        0    l1_cmd:       0
                                       | l2_pso:        0    l2_cmd:       0
 error:         0                      | l3_pso:        0    l3_cmd:       0
                                       | l4_pso:        0    l4_cmd:       0
---------------------------------------+---------------------------------------
ESTOP 0=D,1=P,2=R                 0    |SKYNET INTERFACE
 st delay: 250000  about:          0   |
 status_e:      0  command_e:      0   |Skynet Key: 47576
 status:        0  superCon:       2   |Gas cmds:       0  trans cmd:      0
 position:      1  darpa:          1   |Steer cmds:     0
 error:         0  astop:          1   |Brake cmds:     0
---------------------------------------+---------------------------------------
OBDII                                                                        0
Status_e:      0
Status:      0   st delay: 250000   cm delay: 100000
Engine : Speed:  0.0       Wheel Force:     0.0
SLOW UPDATES:: RPM:    0     Temp:   0.0
-------------------------------------------------------------------------------
SUPERVISORY  delay: 1000000  interlocks on:      1                           0
===============================================================================
Press to send comand|g:gas, s:steer, b:brake, c:acceleration, v:velocity, t:trans.
Press e to reset steering after a disable and re-enable.

Error codes are: 0 is good,  >= 1 is bad
```

Figure 2: This is a screen shot of the Sparrow user interface[7] for the module Adrive. This shows the 6 actuators Steering, Gas, Brake, Transmission, Estop and OBD-II and their status. In this case everything is disabled. Also statistics for the network interface and supervisory thread are displayed.

For the two sensors that did not need commands, the implementation was exactly the same except only the status thread was run.

The network threads consisted of a listening thread and an broadcasting thread. The listening thread opened a socket and listened for incoming commands. When a command was received the cached value for the latest command was updated and a pthread broadcast flag was send to wake the appropriate actuator command thread. However this was conditioned on a set of rules for incoming commands, if the conditions were not met the command would be dropped. An example of one of these rules was that to accelerate, the brake must be off, the car in drive or reverse, and the engine running. The broadcast flag causes that actuator's command thread to wake and execute the latest command that was just recorded. After checking the rules and waking the actuators command thread the network listening thread returned to waiting for an incoming command. The broadcasting thread periodically reads the current cached state of each actuator and broadcasts the values across the network.

For a user interface a thread was created that printed to the screen the states of all the actuators. The ability to set commands was also possible. It was an implementation of the

Sparrow user interface[7]. The other function was that keys were bound that allowed the pthread conditional flags to be broadcast manually. The interface can be seen in Figure 2.

Lastly the supervisory thread took advantage of the locally stored actuator state data to monitor status. The thread checked the status of each actuator periodically for bad performance or failure. If either was found, the supervisory thread would report to the vehicle level contingency management that there was an actuator error. After which the supervisory thread would progress through a series of recovery procedures based on what type of error was detected in an attempt to recover. A simple example of this is that if the engine stalled, attempts to drive forward would be stopped, the vehicle would be put into park, then the starter motor run, and the car put back into gear, before normal operation would resume.

With the 14 threads described above four commanded actuators and two sensors can be interfaced to control over a network. The other five threads provided auxiliary functionality such as logging in a number of different forms.

5 Results

The module, Adrive, has proven to be a successful means of controlling an autonomous vehicle. Adrive has logged over 300 miles of completely autonomous driving.[6] The implementation is scalable to more actuators. This can be seen in Figure 3. Furthermore as can be seen in Figure 4, the computational intensity is very low and does not significantly increase even when commands are received at much higher rates than are necessary for successful control. Also the memory usage is below four megabytes. These computational resources can be provided by most modern computers.

The control performance was observed to be degraded by less than 0.01Hz. This is based on the measured average delay in execution of 25.9 microseconds between receiving a command and sending the command to the actuator. Which, with a 100Hz command rate and a 10Hz maximum operating frequency for the actuator, yields a 9.0888 Hz effective bandwidth. This is in comparison to the theoretical maximum of 9.0909 Hz. Thus the speeds of command response were effectively limited only by the serial communication speeds. However, there was one area where field testing led to modifications of the software.

The greatest improvement made to the software was to implement a rate cap on the speed at which commands are executed. It was observed that commands were not being executed with a lag on the order of 200ms. Receiving commands at 100Hz from the follower means that there is only 10ms between each command, however the communications over a serial port take on the order of 100ms. By implementing a maximum rate limiting for the command thread at 10Hz it eliminated the occasional 200ms delays. The delay occurred when a new command was received and began execution and while executing the status thread ran. The status thread would block and wait for the command thread to execute. When the command thread finished executing, the status thread would assume control of the port and the next command would block 10ms later. The command thread would continue to block until the status thread finished. Then the command thread would execute the previously cued value.

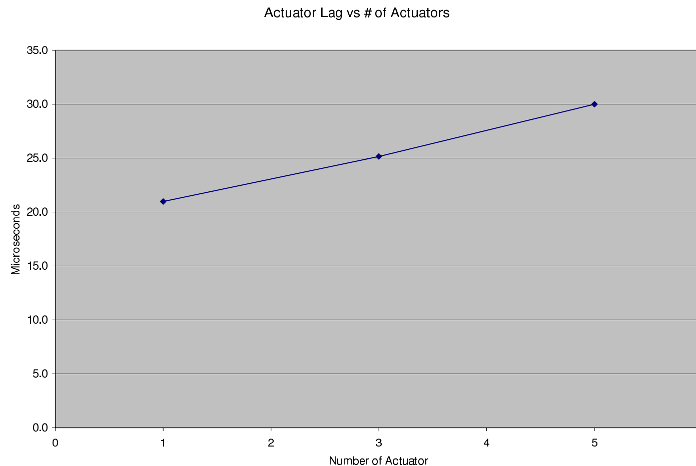

Figure 3: This is a plot of the average delay between receiving a command and sending it out over the serial port as a function of the number of actuators running.

Thus a 50ms delay would be turned into 150ms by cuing the commands. The 100ms delay prevented the accumulation of cued commands but also decreased the potential performance.

6 Conclusion

The implementation described in this paper has been shown to be scalable in both the number of actuators as well as in command rate. With a large bandwidth differential the asynchronous method approaches the theoretical optimum. Using open source software and consumer computer hardware a relatively adaptable and scalable architecture that allows many asynchronous control interfaces simultaneously can be written and work well. Using this implementation Alice has driven hundreds of miles across desert terrain at speeds up to 15m/s.

However there are many areas in which this design could be improved. First of all the code was all written in C with the pthreads library. However the application is very structured and would become very adaptable if it was rewritten into an object oriented structure. The optimal solution would be to have a vehicle class that contains actuator classes each of that are inherited from a basic actuator class with all the basic assessors and initialization function calls built in. Inside each actuator class would be all the necessary drivers as well as the threads and initialization functions. With good abstraction from the software side all actuators would look exactly the same.

A second way to greatly improve the performance would be to put Kalman estimators into the code instead of just caching the last reported value. Using a Kalman estimator, the frequency of status checks could be dynamically lowered based on the confidence of the

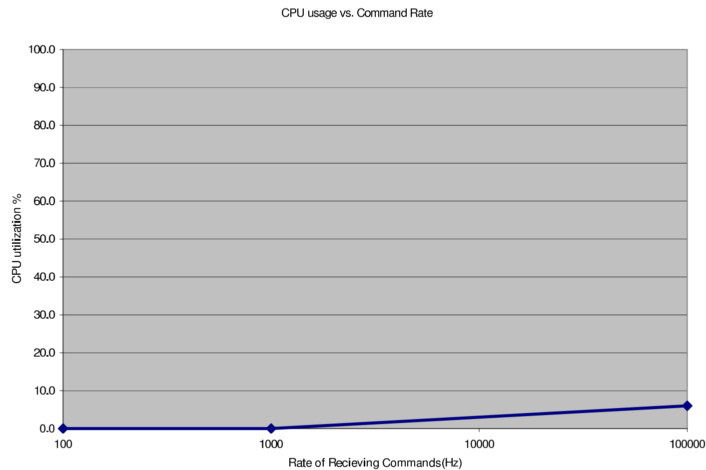

Figure 4: This is a plot of the CPU utilization for Adrive as a function of the number of commands sent over the network per second. 100 Hz was the speed used for the race.

estimate. Decreasing the frequency of the status checks would allow greater bandwidth for the commands to go through.

Lastly, a resolution should be found to the cued command problem discovered while testing. Although a solution was found to prevent the excessive delays it hurts the average performance that does not need to happen.

7 Acknowledgments

All the members of Team Caltech were instrumental in helping test and develop the module as a part of the complete platform. We would like to especially acknowledge the guidance of Richard Murray, Joel Burdick, and Ben Brantley, and the help of Ike Gremmer.

References

[1] Defense Advanced Research Projects Agency. DARPA Grand Challenge 2004. http://www.darpa.mil/grandchallenge04/.

[2] Defense Advanced Research Projects Agency. DARPA Grand Challenge 2004. http://www.darpa.mil/grandchallenge05/.

[3] Yair Amir, Michal Miskin-Amir, and Jonathan Stanton. The Spread Toolkit. http://www.spread.org.

[4] Team Caltech. Team caltech sponsors. http://team.caltech.edu/sponsors.html.

[5] Team Caltech. Team Caltech WIKI. http://gc.caltech.edu/wiki.

[6] Cremean et al. Alice: An information-rich autonomous vehicle for high-speed desert navigation. *Journal of Field Robotics*, 2005.

[7] Richard M. Murray. Sparrow primer. Division of Engineering and Applied Science California Institute of Technology, 1995.

[8] NASA. Mars exploration rover mission. http://marsrover.nasa.gov/.

[9] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly, 1998.

[10] Michael J. Pont. *Embedded C*. Pearson Education Limited, 2002.