Techniques for Web Telerobotics

Barnaby Dalton

This thesis is presented for the degree of Doctor of Philosophy

Department of Mechanical and Materials Engineering
University of Western Australia
Perth, Australia

2001
Abstract

Web Telerobotics is a new and rapidly growing field of telerobotic research. Initially it was considered a novelty, but developments in the field combined with rapidly improving Internet technologies have opened up new application areas. The most promising area is the use of web devices as teaching aids in school and university course work.

Given that web telerobots are appearing in increasingly large numbers with many varying applications, the motivation for this research was to identify, implement, and evaluate techniques for controlling robots (and other devices) over the web. Of particular importance was the identification of a set of requirements that all web telerobot applications require. A solution to these requirements could then be encapsulated in a form that is easily reusable in other web control projects.

In order to understand what was required in a telerobot system, two systems were built. The first used the common gateway interface and enabled operators to control the robot using a standard web browser. During the course of operation and development of this system it became apparent that some functionality was impossible to implement due to the underlying techniques that had been used. This led to the identification of a set of functional requirements that should be provided by any web telerobotics framework.

A Java based framework was developed to implement these requirements. The framework consisted of a central router (MOM) that clients connected into. Users and applications connected via the MOM were able to collaborate together to achieve tasks using shared objects known as contexts. There are three types of contexts: domains – used to define shared areas, channels – used to exchange messages of a particular type, and tokens – used to control access to resources. The framework was designed to be generic and reusable in other web applications.

The framework not only enabled device control over the Internet, but also opened up new possibilities for distributed control. Multiple operators can cooperate together, while software agents contribute to analysis and planning. Multiple robots can be controlled by one operator, or a user could use multiple interface screens to control a single robot. In fact, applications of this framework are
not limited to control of robots. It is suitable for any application that requires collaboration and control over the Internet.

The framework was tested by applying it using the original telerobot hardware. The final system consisted of a number of servers that controlled hardware (such as cameras, and the robot), helper applications, and any number of Java Applets operated by users. The framework enabled all these different applications to communicate and collaborate together, and to negotiate for control of the robot. The system has run for the last year and has had thousands of users. Analysis of usage statistics and user feedback has shown that it is an improvement on the original common gateway interface system, and has shown users collaborating together in ways that were not previously possible.
# Contents

Abstract iii  
Contents v  
Figures and Tables ix  
Acknowledgements xiii  

1 Introduction 1  
1.1 Experimental Setup 4  
1.2 Scope of this thesis 5  
1.3 Development Timeline 5  

2 Background and Related Work 7  
2.1 Telerobotics 7  
2.2 Web Telerobots 8  
2.2.1 Web Interface for Telescience (WITS) 9  
2.2.2 Berkeley 11  
2.2.3 Xavier 12  
2.2.4 Pumapaint 14  
2.2.5 Swiss Federal Institute of Technology 16  
2.2.6 TeleZone 18  
2.2.7 Other Web Robots 20  
2.3 Other telerobots 21  
2.3.1 Distributed Telerobotics 23  
2.4 Summary 25  

3 The CGI System 27  
3.1 Extending HTTP and HTML 27  
3.2 CGI 28  
3.2.1 WIN CGI 30  
3.2.2 Generating Data for CGI 30  
3.2.3 State Management 32  
3.2.4 Other Techniques for producing dynamic content 33  
3.2.5 Frames 34  
3.3 Client Side Extensions 35  
3.3.1 Scripting languages 35  
3.3.2 Plugins 35  
3.3.3 Executables 36  
3.4 Using CGI to control a Robot 36  
3.5 The UWA System 37  
3.5.1 Architectural Changes 38  
3.5.2 Functional Changes 40  
3.6 CGI Implementation 44
6.3.3 Message Oriented Middleware .............................................. 108
6.3.4 Comparison of Middleware .................................................. 108
6.4 The Framework ................................................................. 109
  6.4.1 The Router ................................................................. 109
  6.4.2 Peers ................................................................. 111
6.5 The Framework in Practice ..................................................... 112
6.6 Future Work and Conclusion ................................................... 115

7 Distributed Architecture .......................................................... 117
  7.1 Revised Framework .............................................................. 119
    7.1.1 Contexts ............................................................... 119
    7.1.2 Transceivers .......................................................... 121
    7.1.3 Parcels ............................................................... 122
  7.2 The MOM Architecture ......................................................... 122
    7.2.1 Communication Layer .................................................. 122
    7.2.2 Context Layer ........................................................ 124
    7.2.3 Parcels ............................................................... 125
    7.2.4 Queues ............................................................... 127
  7.3 Peer Architecture .............................................................. 128
    7.3.1 Communication Layer .................................................. 129
    7.3.2 Context Layer ........................................................ 130
    7.3.3 Client Layer .......................................................... 132
    7.3.4 Presentation Layer ..................................................... 132
  7.4 Communication ................................................................. 132
    7.4.1 Socket Transceiver ..................................................... 133
    7.4.2 RMI Transceiver ....................................................... 139
    7.4.3 Other Transceivers ..................................................... 142
    7.4.4 Comparison of Transceivers ......................................... 143
    7.4.5 Adaptive Bandwidth and Stream Separation ......................... 144
  7.5 Source Code Example .......................................................... 145

8 The Framework in Practice ....................................................... 149
  8.1 MOM Based Telerobot System ................................................. 149
  8.2 Robot Subsystem ............................................................. 150
    8.2.1 Robot Server ......................................................... 150
    8.2.2 Robot Clients ......................................................... 152
  8.3 Image Subsystem ............................................................. 153
    8.3.1 Image Server ......................................................... 153
    8.3.2 Image Client ......................................................... 158
  8.4 Peer Collaboration ........................................................... 159
    8.4.1 Chat ................................................................. 160
    8.4.2 Workspace Modelling ................................................ 160
  8.5 Client Applet ................................................................. 162
    8.5.1 The Image Panel ...................................................... 163
    8.5.2 Robot Command Panel ............................................... 170
    8.5.3 Console Panel ........................................................ 172
    8.5.4 Using the Applet ...................................................... 172
  8.6 Results and Experiences ..................................................... 173
    8.6.1 Use of the system ..................................................... 174
    8.6.2 User Collaboration .................................................... 179
8.6.3 User Feedback .............................................. 181
8.6.4 Applet practicalities ..................................... 184
8.7 Calibration Application .................................. 187
8.8 UWA Telelabs ................................................. 189
  8.8.1 Applying the framework ............................... 190

9 Discussion ......................................................... 193
  9.1 Summary ...................................................... 193
  9.2 Contributions .............................................. 195
  9.3 Conclusions ................................................ 196
  9.4 Future Work ................................................. 197

Bibliography ......................................................... 201

Glossary ............................................................. 211

A Related Publications ........................................ 215

B Computer Vision .............................................. 217
  B.1 Elements of the Camera Matrix ...................... 217
  B.2 Calculation of Pose Parameters ................. 218

C Chat Conversations and Comments ...................... 221
  C.1 Selected Chat conversations ..................... 221
  C.2 Selected Comments .................................. 225
Figures and Tables

List of Figures

1.1 Two views of the experimental setup .................................. 4
1.2 Timeline for software development ...................................... 6

2.1 WITS panorama .......................................................... 9
2.2 WITS mosaic ............................................................. 10
2.3 The Telegarden ........................................................... 12
2.4 Xavier observation page ............................................... 13
2.5 Xavier control page ..................................................... 14
2.6 The Pumapaint interface .............................................. 15
2.7 Khep on the Web ......................................................... 16
2.8 The RobOnWeb interface ............................................. 18
2.9 The TeleZone ............................................................ 19

3.1 Example HTTP GET request ........................................ 28
3.2 Example CGI POST request ........................................ 29
3.3 A reply from a CGI script ........................................... 29
3.4 Windows ini file used to launch the telerobot CGI program ...... 31
3.5 Original CGI Architecture ........................................... 37
3.6 Modified CGI Architecture ........................................ 39
3.7 An early HTML interface ............................................. 41
3.8 Different zoom settings ............................................... 43
3.9 Schematic diagram of CGI system ................................... 44
3.10 The global status file ................................................ 45
3.11 State transition diagram for users ................................ 46
3.12 Activity diagram for the CGI script ............................... 48
3.13 Spin and tilt orientations ........................................... 49
3.14 Multiple move syntax ............................................... 51
3.15 Timing diagram for synchronous images ......................... 53
3.16 Timing diagram for asynchronous images ....................... 53
3.17 State information paths ............................................. 54
3.18 Raw HTML Template ................................................ 54
3.19 Types of HTML template tags ..................................... 55
3.20 Tags used by CGI script ............................................ 56
3.21 The default user interface ......................................... 57
3.22 Robot server control flow ......................................... 57
3.23 Image server hardware ............................................ 61

4.1 Number of hits per week on different parts of the CGI telerobot system ............ 64
4.2 Histogram of moves in a sessions .................................. 65
4.3 Histograms of times between moves ................................ 66
4.4 Distance moved per week from January 1997 to April 1998 ......................... 67
4.5 Histogram of robot poses moved to by operators ................... 68
4.6 A selection of the more accomplished creations ...................... 69
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7</td>
<td>Types of error during a move</td>
<td>70</td>
</tr>
<tr>
<td>5.1</td>
<td>Conceptual layers of TCP/IP</td>
<td>79</td>
</tr>
<tr>
<td>5.2</td>
<td>The windowing protocol used by TCP/IP</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>Time taken to establish a connection between a client and server</td>
<td>87</td>
</tr>
<tr>
<td>5.4</td>
<td>The effect of application message size on round trip delay</td>
<td>88</td>
</tr>
<tr>
<td>5.5</td>
<td>Effect of packet size on round trip time for UDP packets</td>
<td>90</td>
</tr>
<tr>
<td>5.6</td>
<td>Distribution of failure rate for UDP packets</td>
<td>91</td>
</tr>
<tr>
<td>5.7</td>
<td>The effect of packet size on failure rate</td>
<td>91</td>
</tr>
<tr>
<td>5.8</td>
<td>Network characteristics for different hosts</td>
<td>92</td>
</tr>
<tr>
<td>5.9</td>
<td>Network characteristics for different hosts</td>
<td>93</td>
</tr>
<tr>
<td>5.10</td>
<td>The effect of buffering data</td>
<td>95</td>
</tr>
<tr>
<td>5.11</td>
<td>Peer to Peer connections</td>
<td>102</td>
</tr>
<tr>
<td>5.12</td>
<td>Peer to router architecture</td>
<td>103</td>
</tr>
<tr>
<td>6.1</td>
<td>Peer to router architecture</td>
<td>109</td>
</tr>
<tr>
<td>6.2</td>
<td>Message structure for sent and received messages</td>
<td>110</td>
</tr>
<tr>
<td>6.3</td>
<td>The architecture of a peer</td>
<td>111</td>
</tr>
<tr>
<td>6.4</td>
<td>The peer and router architectures</td>
<td>113</td>
</tr>
<tr>
<td>6.5</td>
<td>Demonstration Applet controlling a IRB 2400 robot</td>
<td>114</td>
</tr>
<tr>
<td>7.1</td>
<td>Contexts organised in a tree structure</td>
<td>121</td>
</tr>
<tr>
<td>7.2</td>
<td>UML Model showing the Listener interface</td>
<td>124</td>
</tr>
<tr>
<td>7.3</td>
<td>The various Parcel classes</td>
<td>126</td>
</tr>
<tr>
<td>7.4</td>
<td>MOM Queues and threads</td>
<td>128</td>
</tr>
<tr>
<td>7.5</td>
<td>The layers of the MOM architecture for a robot client</td>
<td>129</td>
</tr>
<tr>
<td>7.6</td>
<td>UML model showing the peer proxy contexts hierarchy</td>
<td>131</td>
</tr>
<tr>
<td>7.7</td>
<td>The transceiver interface</td>
<td>134</td>
</tr>
<tr>
<td>7.8</td>
<td>The Socket Transceiver</td>
<td>135</td>
</tr>
<tr>
<td>7.9</td>
<td>Interaction between the socket transceiver</td>
<td>136</td>
</tr>
<tr>
<td>7.10</td>
<td>Class diagram of XML and Java translators</td>
<td>137</td>
</tr>
<tr>
<td>7.11</td>
<td>An example of a parcel serialised as XML</td>
<td>138</td>
</tr>
<tr>
<td>7.12</td>
<td>The RMI Transceiver object hierarchy</td>
<td>140</td>
</tr>
<tr>
<td>7.13</td>
<td>Steps required to setup a connection using RMI</td>
<td>141</td>
</tr>
<tr>
<td>7.14</td>
<td>Source code for the chat log peer</td>
<td>146</td>
</tr>
<tr>
<td>8.1</td>
<td>The underlying architecture used to represent cameras</td>
<td>156</td>
</tr>
<tr>
<td>8.2</td>
<td>Java architecture of image server</td>
<td>157</td>
</tr>
<tr>
<td>8.3</td>
<td>Code for sending a new image to a peer</td>
<td>158</td>
</tr>
<tr>
<td>8.4</td>
<td>Client code to change a camera’s settings</td>
<td>159</td>
</tr>
<tr>
<td>8.5</td>
<td>Code to send a chat message</td>
<td>161</td>
</tr>
<tr>
<td>8.6</td>
<td>The Applet interface used to move the robot</td>
<td>162</td>
</tr>
<tr>
<td>8.7</td>
<td>The image panel</td>
<td>163</td>
</tr>
<tr>
<td>8.8</td>
<td>Elements of the stick cursor</td>
<td>165</td>
</tr>
<tr>
<td>8.9</td>
<td>Dial plates shown for spin and tilt orientation</td>
<td>165</td>
</tr>
<tr>
<td>8.10</td>
<td>The stick cursor</td>
<td>166</td>
</tr>
<tr>
<td>8.11</td>
<td>Object identification in progress</td>
<td>170</td>
</tr>
<tr>
<td>8.12</td>
<td>The robot command panel</td>
<td>171</td>
</tr>
<tr>
<td>8.13</td>
<td>The number of hits per week for different parts of the Web site</td>
<td>175</td>
</tr>
<tr>
<td>8.14</td>
<td>Distance moved by the gripper in a week, using the MOM system</td>
<td>176</td>
</tr>
<tr>
<td>8.15</td>
<td>Number of moves in a session for the new MOM system</td>
<td>176</td>
</tr>
</tbody>
</table>
LIST OF TABLES

8.16 Histograms of the time between moves .......................... 177
8.17 Histogram of robot poses moved to by operators .................... 178
8.18 A short investigative session by a user .......................... 180
8.19 Questionnaire results ................................................. 183
8.20 The (Java) calibration application ..................................... 188

B.1 Techniques used to calculate new pose values ......................... 219

List of Tables

3.1 CGI output headers ..................................................... 30
3.2 Comparison of server side extensions .................................. 34
5.1 Connections made from remote hosts .................................. 86
5.2 Results of UDP test Applet ........................................... 89
8.1 Types of gripper states at the end of a robot move .................... 179
Acknowledgements

I would like to thank my supervisor, Associate Professor James Trevelyan, for his guidance throughout the project, and for his faith and patience in what has been a rather protracted writing up period.

This work stems from initial work in web telerobotics by Dr. Kenneth Taylor, which included the original software that ran the robot first connected up to the web 1994. Much of my initial work was done in close cooperation with Ken, who had many interesting ideas and experiences from the first few years of operation over the Web.

During the project there were many helping hands. These included: Gintarus Radzavinas who implemented the multiple move script, Stephen LePage who developed the user registration, and Harald Friz, an exchange student who designed the first Java interface that included the augmented reality stick cursor. Thanks must also go to the workshop team who were called upon whenever the trials of online operation proved too much for the robot (or the table).

Deserving special thanks is Sabbia Tilli, who I shared an office with and who was always there to help. She has also kept the robot running for the last year in my absence. A special thanks too to my proof readers Charlotte Bradshaw who gave up time and energy during her maternity leave, and my wife, Tara who has lived graciously with this thesis for a long time.

Finally, a thank you to the thousands of people who used the robot, providing valuable feedback, motivation and statistics on how to improve the system.
Web telerobotics started in 1994 with Goldberg’s Mercury project (Goldberg et al., 1995a), and Taylor’s Australia’s Telerobot project (Taylor and Trevelyan, 1995). It has since grown in size with many web telerobots now online. There is at least one company commercially exploiting the technology\(^1\), and it is beginning to find applications in areas such as education, space teleoperation (Backes et al., 1998) and entertainment.

Australia’s telerobot and the Mercury project were not the first robots to be controlled over the Internet. As early as the early 1990’s the Internet had been used in telerobot control experiments (Kondraske et al., 1993; Kim and Bejczy, 1993; Stein, 1994). However, interestingly these experiments were not motivated by the growing reach and capabilities of the Internet but by its delay characteristics which provided a cheap test environment for simulating time delayed space teleoperation.

Web telerobots were different in that they were made freely available using the open web standards (HTTP and HTML). The required client software (a web browser) was already installed on a user’s machine. No additional software or hardware was required, so that anybody anywhere could control a robot almost immediately with no setup or installation required. It was this fact that made them so different to previous closed, and proprietary telerobotic systems. The history and potential applications of telerobotics and more recently web telerobotics is discussed in chapter 2.

Given that web telerobots are appearing in increasingly large numbers with many varying applications, the motivation for this research was to identify, evaluate, and implement techniques for controlling robots (and other devices) over the web. Of particular importance was the identification of a set of requirements that all web telerobot applications require. A solution to these requirements could then be encapsulated in a form that is easily reusable in other web control projects.

\(^1\)Perceptual Robotics has been founded to exploit the web teleoperation of cameras.
Research for this thesis started in 1996 at the University of Western Australia(UWA), working with Ken Taylor on the completion of a port of Taylor’s original telerobot system to work with a new robot (an ABB 1400 manipulator), and a new operating system (Windows 3.11). Taylor’s original system which ran under DOS was no longer operational as the original robot arm had been sold. The system used the common gateway interface (CGI) to generate web pages that allowed users to control the robot. A number of issues were identified with the original system which included: the time taken to process requests, reliability, and lack of flexibility. Additionally, in the hope of identifying reusable parts, the system needed to be split into logical subsystems. These requirements led to a partitioning of the system into a number of separate applications that communicated with each other. These included the CGI process, a robot server process, and an image server process. This not only partitioned the system, but also improved its speed. Taylor’s thesis (Taylor, 1999) contains a comparison of the original system and Windows 3.11 which shows that the new version was a significant improvement from the user’s perspective. As part of this initial work an HTML template library was also developed to allow different user interfaces to be designed without requiring code recompilation. This not only allowed different designs to be tried locally but also allowed users to design their own. Both the template library and image grabbing server were reused in Carter’s Lab Cam project (Carter, 1997). Chapter 3 discusses the CGI techniques used in the UWA implementation.

The Windows 3.11 CGI system came online in late 1996, and continually evolved with new features, such as a multiple move script, user logins, and new interface designs. The results of these changes and how they affected usage of the robot are presented in chapter 4. However, with these improvements came a growing realisation of the limits of an HTTP/CGI interface combined with an HTML interface.

Some of the limitations of HTTP/CGI include: the lack of server initiated communication, unreliability in user session management, problems managing state, and the difficulty of performing incremental updates in an environment where the fundamental unit of communication is a document. Some of these issues can be addressed with HTTP extensions such as cookies, and server push but the system is still nominally constrained by the original intentions of HTTP - to send and receive hypertext pages, not to control robots or other devices!

The shortcomings of HTML include the limited user interface toolkit, and the page based interface imposed by a browser. To an extent these can be overcome with extensions to HTML such as frames, Javascript and browser plugins. However, these still result in an interface that must conform to a page based environment.

---

2 With the exception of plugins, but these have their own associated problems.
Java Applets provide the capability to overcome some of the limitations of HTML and HTTP. Harald Friz (Friz, 1998) experimented with a Java control interface which proved its capabilities as a replacement for HTML, while still using HTTP and the CGI script for communication and control. To test the network capabilities and performance available to a Java Applet, a number of test Applets were deployed and run by hundreds of users across the web. These gave an indication of what network operations were possible, and what sort of network performance to expect. These network tests and results are the subject of chapter 5.

After investigating various network protocols and technologies, a Message Oriented Middleware (MOM) based approach was adopted to develop a reusable framework that could not only be used for the UWA Telerobot but also be applied to other web controlled devices. This framework extended the system from being a single user, single robot system into a multiuser, concurrent distributed system. This ability for the system to be distributed is perhaps the most promising area of web telerobotics. The traditional view of a telerobot consisting of a robot and remote operator can be replaced by multiple robots, multiple users, and maybe even other autonomous agents all geographically distributed, with all peers in the system being connected via the Internet and therefore able to communicate and collaborate with each other. Although there are many distributed systems frameworks available, none were found to be easily usable from within a Java Applet. To work well within an Applet, a framework needs to be available in a Java form, freely distributable (no license fees), small (10s of KBytes), and amenable to network constraints (security restrictions and firewalls). It must also be able to work reliably with the widely differing network connection speeds of end users and should address issues such as user authentication, resource and access control, and synchronous and asynchronous communication. If a framework is able to address these issues and provide the necessary services, then new applications which use the framework only need to implement functionality specific to the problem they are solving. The design and implementation of such a framework is presented in chapters 6 and 7.

The framework was applied to the UWA Telerobot to produce a multiuser collaborative system in which users could talk and help each other, observe the robot, negotiate for control, and add points to a shared model of the robot workspace. The user interface was a Java Applet that included an augmented reality image of the workspace that could be used for modelling and understanding the remote environment. Users were all permanently connected to a central server known as the MOM. As each client was permanently connected, events were propagated as soon as they occurred. This MOM system operated for a year from March 2000 to March 2001, without any intervention by the author. The details of the implementation and results of the year’s operation are discussed in chapter 8. An unexpected application of the framework was that it made integration of a calibration program into the system very easy. This calibration application could be run from
anywhere in the same way as the normal Applet interface, and meant that the author was able to calibrate the robot in Australia while working in London.

1.1 Experimental Setup

Although the software setup changed significantly, the physical robot setup remained largely unchanged for the duration of this work. Figure 1.1 shows two photographs of the robot lab. The robot arm is positioned above a table on which there are wooden blocks. A number of cameras are placed around the table to provide feedback to operators. By moving the robot and opening and closing the gripper, users could manipulate the blocks to create (or more frequently destroy) structures. Stacking of blocks was chosen as the task to be used to evaluate the system’s performance. It is a semi structured task that is hard be perform autonomously, and therefore requires a human tele-operator.

Figure 1.1: A view of the robot layout. The arm is positioned over a table with blocks on it. Cameras are placed around the table providing visual feedback to operators.

It was a standard industrial robot arm (an ABB 1400), controlled via an ABB S4 controller. The robot was equipped with a pneumatic gripper that could be opened or closed. Although no force sensors were used, the robot had a current overload detector on each joint which provided a very crude form of collision detection. In late 1999 a proximity sensor was added to the gripper to detect the presence of objects between the jaws. User feedback was provided by a number of cameras, with the position and number of cameras varying from 2 to 4 during the course of the work. A colour camera was included in the final Java system.
1.2 Scope of this thesis

Telerobotics systems can consist of complex subsystems at both the remote site, and the operator station. The space teleoperation systems funded by NASA (Backes et al., 1991; Bejczy et al., 1994; Backes et al., 1995) and the European space agency (Hirzinger et al., 1989, 1993) represent many man years of work in developing all aspects of a telerobot system. This includes improved sensing, error recovery, control and ultimately intelligence for the remote site, and includes improved feedback and visualisation and command language on the operator site. Each of these areas are research areas in their own right.

To avoid repetition of previous telerobotics work, this work concentrates on the aspects of telerobotics specific to web telerobotics. This includes: the communication protocols, distributed systems issues such as user authentication and resource control, and the overall architecture of the system. It also includes the user interface, as this is constrained to operate within the confines of a standard computer hardware - a mouse keyboard and monitor.

As the research was into the web aspect of web telerobotics the robot and its sensors were treated as a black box, and they were not developed in any way. The S4 controller provided a programming language interface to the robot. The language allows paths and waypoints to be specified but allows little lower level intervention. This means that compared to the telerobots mentioned above, the robot and its sensors were relatively simple. However, local control aspects are essentially unrelated to the web specific aspects mentioned above, so the web techniques discussed in this thesis can be applied to a more complex telerobot setup.

1.3 Development Timeline

Previous to this work, Ken Taylor’s system had used an IRB6/2 robot which ran under the DOS operating system (this was the original telerobot and is referred to as the Taylor96 system). Figure 1.2 shows the period of development for all the systems described in this thesis. Initial work in 1997 consisted of completing a port of the existing system to work with the ABB 1400 robot, and the Windows 3.11 operating system. This was followed by various enhancements to the robot aimed at making it easier to use. These included a zoom feature, the use of HTML templates, and feedback of robot errors to operators. These are discussed in chapter 3. In 1998 the system was again ported, this time to work under the Windows NT operating system which offered far greater reliability. Socket communication was introduced so that different parts of the system could be on different machines, users were given access levels via a login page, and a script language was
introduced to program multiple moves. These are also discussed in chapter 3. In late 1997 Harald Friz started work on the user interface which was the first version of the Augmented Reality cursor described in chapter 8. This interface still used the CGI script to control the robot.

From early 1998 work started on the distributed architecture of chapters 6 to 8. The first version talked about in chapter 6 was finished in late 1998. Due to major building work at the department from the middle of 1998 to early 1999, the robot was moved to Questacon, a science museum in Sydney. During this time it was still running the CGI system, but due to some problems at the installation site, the robot was off-line for most of this time. Most of 1999 was spent developing a second version of the framework, an initial version of which was tested while the author worked with ABB research in Norway. The final version finally came online in early 2000. At this point the author left Australia to take up a job in London. The system was maintained by James Trevelyan and Sabbia Tilli until March 2001, the period of time used for the results in chapter 8.

Figure 1.2: Timeline for software development

The Thesis development Timeline

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CGI</td>
<td>Socket Communication</td>
<td>Questacon error</td>
<td>MOM Mark II</td>
</tr>
<tr>
<td>Windows 3.11 Port</td>
<td>User Login</td>
<td>Questacon</td>
<td>Norway MOM Prototype</td>
</tr>
<tr>
<td>Software Zoom</td>
<td>Multiple Moves</td>
<td>Router Connector</td>
<td>MOM</td>
</tr>
<tr>
<td>Templates</td>
<td>NT Port</td>
<td>MOM Applet</td>
<td>Final MOM System</td>
</tr>
<tr>
<td>Robot Errors</td>
<td>Usher Applet</td>
<td>MOM Router</td>
<td></td>
</tr>
</tbody>
</table>

From early 1998 work started on the distributed architecture of chapters 6 to 8. The first version talked about in chapter 6 was finished in late 1998. Due to major building work at the department from the middle of 1998 to early 1999, the robot was moved to Questacon, a science museum in Sydney. During this time it was still running the CGI system, but due to some problems at the installation site, the robot was off-line for most of this time. Most of 1999 was spent developing a second version of the framework, an initial version of which was tested while the author worked with ABB research in Norway. The final version finally came online in early 2000. At this point the author left Australia to take up a job in London. The system was maintained by James Trevelyan and Sabbia Tilli until March 2001, the period of time used for the results in chapter 8.
CHAPTER 2

Background and Related Work

Web telerobots use standard web technologies such as HTTP, HTML, and Java\(^1\) to provide a telerobotic control interface over the Internet. The first web telerobots came into existence in 1994. These were the robots of Goldberg (Goldberg et al., 1995a) and Taylor (Taylor and Trevelyan, 1995). Since then the number of web telerobots has been growing rapidly. Although initially considered a novelty, improvements in the underlying Internet technology have meant that more and more devices are being controlled or monitored via the web. This chapter documents some of the different web telerobots that have been developed and discusses how they have been implemented.

2.1 Telerobotics

The field of telerobotics grew out of the need to perform operations where it is difficult to place a human being, due to constraints such as cost, safety or time. Telerobotic systems need to be able to perform tasks that a human would normally do. Due to limitations in robot autonomy, this often has to be achieved by using human operators to control the remote robot (via a communication link). Such a system is a telerobot. The human operator is responsible for high level control such as planning and perception, while the robot performs the low level instructions at the remote site. For an in depth coverage of the field of telerobotics (see Sheridan, 1992).

An important property of a telerobotic system is the method by which it is controlled. Conway (Conway et al., 1990) defines the following classification of telerobotic control schemes:

**Direct continuous control**  The remote device follows the inputs from the controller; also known as master/slave control.

\(^1\)For definitions of these terms please see the next chapter and the glossary.
**Background and Related Work**

**Shared continuous control** Control is at a higher level than direct position servoing; i.e. the device may vary from course if it encounters an obstacle.

**Discrete command control** The controller is able to carry out discrete commands without intervention. This implies a higher level of capability in the remote portion of the controller as it must be able to carry out the command without help.

**Supervisory control** The remote device operates in a largely autonomous mode and only interacts with the human when it encounters an unexpected situation.

**Learning control** The remote device is given intelligence that allows it to learn from human inputs and sensor information and subsequently to generate behaviour in similar situations without human intervention.

An aspect of web telerobot systems that affects the choice of control scheme is time delay. The Internet protocols do not guarantee a maximum delay for a message to be carried across a network link, which means that the control scheme must work under variable (and possibly large) time delays. Continuous control is not well suited, as it is prone to instability problems under time delay (Sheridan, 1993). Shared continuous control is less sensitive to these problems and has been demonstrated over the Internet (Tzyn-Jong and Brady, 1998; Brady and Tzyn-Jong, 1998, 1999), but only on short, high bandwidth Internet links. Discrete command control schemes and above are free of any time delay based instability problems as all closed loop control is performed locally. They are therefore the most appropriate choice for web telerobotic systems.

### 2.2 Web Telerobots

This thesis continues the *Australia’s Telerobot on the Web* project started by Taylor (Taylor, 1999) in 1994. At the start of Taylor’s work there were no web telerobots in existence. By the time his system came online in 1994, Goldberg’s (Goldberg et al., 1995a) Mercury project was also in operation. Taylor’s system allowed users to control a 6 degree of freedom arm located above a table which had wooden blocks placed on it. Users could send move requests to the robot via a web page which included images of robot workspace. The system used the common gateway interface to generate dynamic web pages which showed the new state of the robot after each move was made. The system was extremely popular and can be viewed as one of the pioneering web telerobot systems. A very similar system is now installed in the Carnegie Science museum. The system is discussed in more detail in the next chapter.
2.2 Web Telerobots

2.2.1 Web Interface for Telescience (WITS)

NASA has developed a web interface for Telescience (WITS) (Backes et al., 1997, 1998, 1999; Backes) for controlling remote vehicles on planets such as Mars and Saturn. WITS is being developed for use in the 2003 and 2005 rover missions to Mars for distributed rover command generation. WITS was also available in demonstration from for the 1997 Mars Pathfinder mission. The Pathfinder interface only allowed control and observation of a simulated rover but used real data from the Mars mission.

The client interface uses JAVA, and is launched from an HTML page. The Applets are either downloaded each time a user visits the site or kept on the local machine and automatically updated using Marimba push channel subscription. Due to the size and complexity of the interface, a complete download of the interface can take upwards of ten minutes.

To control the rover the user is presented with a number of views of the surrounding environment, the primary views being an overhead panorama (Figure 2.1) and a horizontal mosaic view (Figure 2.2).

Figure 2.1: Panorama interface for the NASA WITS interface. Height calculated from stereo vision images is represented as different colours.
The panorama view shows an overhead view of the scene. The panorama images might be taken by cameras on a lander or on a rover mast. A colour-coded elevation map is generated from panoramic stereo images and shown with the wedge image areas outlined. Current rover location and any selected waypoints and targets are also shown. The mosaic view is a mosaic of panorama images, giving a view over the landscape in a given direction. The direction can be changed to allow viewing in any direction. The images can be displayed at various scales from full size (1) to one eighth size (1/8).

All these images are calibrated and have range data. This means that each pixel on an image is mapped to a point in 3D space. This provides an easy point and click method for measuring the environment. It is even possible to measure the distance between two points by clicking and dragging.

WITS is a multiuser system. Science tasks are associated with each user and stored centrally within the database. Tasks for the rover to execute are programmed via the image maps. Waypoints are specified by clicking, as are observation points where scientific experiments are to be carried out. Hazards and points of interest can also be labelled. At any point a registered user is able to save all marked points to the central JPL database.
2.2 Web Telerobots

Visual tasks can be compiled down to language instructions which can be edited in situ. These can then be used to drive a simulation of the real rover, or in the future be downloaded to a real Rover via the Database and Mission Planner.

The WITS system was developed as a mission planning tool and its purpose is therefore somewhat different to other web telerobots. It allows multiple scientists to collaborate together on a single mission plan stored in a central database. CGI is used to send requests to this database from the client Applet. The interface is completely decoupled from the actual rover which operates autonomously once it receives its instructions for a given day. This means that the interface does not need to provide image or state updates from the rover as it moves. As CGI is used, the interface is only updated with new information when the users makes a request back to the database.

2.2.2 Berkeley

Goldberg et al. (1995b,a, 2000) have produced a number of Internet controlled devices. The first of these was the Raiders Robot believed to be the first robot controlled via a web browser. The robot was located over a dry-earth surface allowing users to direct short bursts of compressed air onto the surface using the pneumatic system. Thus robot operators could ‘excavate’ regions within the environment by positioning the arm, delivering a burst of air, and viewing the image of the newly cleared region. The system used CGI to interface between the webserver and physical devices. The Raiders project is no longer running.

Their second project is a robotic garden (The Telegarden). Users may plant and water the garden or just look around. Movement of the robot is specified by clicking on a graphical representation of the workspace or images, or by specifying a location by grid reference (see figure 2.3). The interface uses standard CGI with no Java or Javascript. Options such as image size, quality and chat interface are all controlled on a second page, keeping the control page extremely simple. Users are encouraged to register by being given only a few rights as guests. Active participation is then encouraged by further increases in rights with user activity. A user must make 100 requests before being allowed to plant a seed. A random helpful message at the bottom of each page helps to gradually educate the user while keeping the information load small. A camera, removed from the garden is mounted on a pan/tilt module allowing viewing of the garden and surrounding area by simple clicks on the image.

The Telegarden is designed to be simple to operate all options are on a second page and robot movements are programmed by point and click. The environment is however quite restricted. All moves are in a plane parallel to the ground so a click on an image is enough to fully define a move.
12 Background and Related Work

Figure 2.3: Interface for the Telegarden. Moves are made by clicking the graphic or images, or by specifying a grid location.

Actions such as watering and planting seeds are autonomous tasks controlled by binary on/off indicators on the controller’s page.

2.2.3 Xavier

Xavier (Simmons et al., 1999) is a mobile robot at Carnegie Mellon University (CMU). As it is a mobile robot it is powered by batteries, which need to be recharged and are expensive to replace. As a result, the robot is online for only a few hours a day (19:00 to 21:00 GMT). The web interface was conceived as a way of testing navigation algorithms of the robot. Initially intended as a short test period, the project has now been running for 4 years. This is due to the surprising popularity of the project. The main research focus has been the local intelligence of Xavier and not its web interface. Being a mobile robot it is subjected to a number of additional constraints, including: limited bandwidth of radio modems and the problem of power mentioned above. Autonomy of the robot combined with supervisory control helps reduce the bandwidth requirement, but at the same time reduces interactivity. Users apparently prefer ‘hands on’ control, and immediate feedback.
Xavier’s control architecture is the product of a number of research projects, and as a result is quite accomplished. Commands to the robot are at a high level, specified as target locations. All planning and navigation is performed by the system.

Figures 2.4 and 2.5 show the observation and control interfaces for Xavier. The commands are sent from the browser to the robot via a CGI script that runs on a server machine. The CGI script communicates with a task manager that allocates tasks based on a simple resource scheduling algorithm. This machine communicates individual tasks to Xavier.

The Xavier researchers state that the most useful aspect of their web experiment is learning about the reliability of their navigation algorithm:

“Nothing beats having naive users to test a system’s reliability. For example the first day that Xavier was online, the software crashed repeatedly. The reason was that users were requesting Xavier to go where it already was, and we had never tested that capability before.”

Other observations for web based telerobotics include the need for high quality feedback. Users need to know what is happening, especially if they have never used the system before. Limited
Background and Related Work

Figure 2.5: Control page for Xavier. The control page shows the high level of commands. Options are to move to a specific place, and to perform a simple task. Navigation to the place and execution of the task are then scheduled and executed autonomously.

time online is identified as the major problem of Xavier as few visitors get the immediate feedback they expect from the web. twenty four hour presence is stated as an important goal for web based robots.

2.2.4 Pumapaint

The Pumapaint project (Stein, 1998; DePasquale et al., 1997; PumaPaint) arose out of a collaboration between Wilkes University and the University of Wisconsin(UW). The computer science department at UW were considering purchasing a PUMA robot for use in undergraduate courses. For cost and maintenance reasons the faculty decided to attempt to share a robot already installed at Wilkes, by allowing students to access it over the Internet. The software to perform Internet control was written as undergraduate projects and the current publicly available version is a direct result of this collaboration.

The interface shown in figure 2.6 enables users to paint on a canvas at Wilkes University using the Puma robot. Commands are sent via a simple paint style program interface. Results of painting can
be viewed from two different angles in separate windows. The interface and communications are implemented entirely in Java. The communication protocol uses a permanent socket connection and custom protocol to send commands and receive feedback. This means that the server is able to asynchronously contact the operator to update command status. It is a stated design aim of the project to give as much control to the user as possible while assisting them unobtrusively when necessary. Users can queue up as many commands as they like, but can also see the size of the queue to determine how far ahead they are working. Similarly the amount of paint left on the brush is indicated by the amount of colour deposited on the virtual canvas, but it is left to the user as to when to replenish the brush with new paint.

The interface uses the Java 1.0 API, and can therefore be run on most browsers. Java 1.0 was supported from an early stage in web browsers. However, to get the interface to work reliably across many platforms and Java versions, extensive testing had to be performed. New platforms/version combinations produced new unexplained problems. Mostly, these problems were due to inconsistencies in the implementation of the Java virtual machines. The fix for these problems was often a workaround rather than a solution. This highlights one of the main problems of developing Java applications that live up to the 'write once run anywhere' trademark of Java. Added to the Java problems, the implementation of a custom protocol over sockets became a problem with America Online (AOL) proxy servers. The AOL proxy servers often stayed connected long after the client...
had lost interest. An automatic disconnect after a maximum idle period was introduced to solve the problem.

2.2.5 Swiss Federal Institute of Technology

The Swiss Federal Institute of Technology Lausanne (EPFL), have developed a number of Internet controlled mobile robots (Michel et al., 1997; Siegwart and Saucy, 1999). The first of these was Khep on the Web - a robot in a maze. Users can move the robot to try and negotiate the maze. The interface uses many different web technologies including CGI, Java, Javascript, frames, VRML, and server push. Javascript and CGI are used in combination to submit requests to the robot. All controls are visual and clickable as can be seen in figure 2.7.

Figure 2.7: Khep On The Web. This Interface uses many different web technologies including VRML, Java, Javascript, frames, server push and CGI.

These controls do not need to change to reflect the current state of the robot as it is uncalibrated and all moves are relative to the current position. Many other Internet devices operate in an uncalibrated mode. This greatly simplifies the system as the position of the robot is fed back via
images and is left to the user to infer. Operating with an uncalibrated robot means that commands must be relative as it is not possible to request absolute positions.

A Java Applet is used in one of the frames to provide updates on the state of the robot. It tells the user when the robot is moving and when it is stationary. It also controls the time that the user is allowed to use the robot for. Images of the robot are fed back using server push. The server push is continuous even when the robot is not moving. This can result in unnecessary image downloads when nothing has changed in the remote environment.

VRML is used to provide a virtual model of the robot and workspace. This can be controlled in exactly the same manner as the real robot, providing training for inexperienced users and an alternative to the real robot when it is in use by another user. Interaction between VRML and Java provides huge potential for modelling and displaying what is/would be happening with a real robot. The model can be written in Java but all display and rendering processing is handled by the VRML engine.

Khep has another interesting facility: the ability to run code from other sources. LabView and C algorithms can be downloaded to test their ability to navigate the maze. Key parameters in the code can be monitored remotely to assess performance as the program runs. This provides anyone access to a robot to test their work without the expense of investing in a robot themselves. It also ensures that all researchers test their algorithms in the same circumstances, ensuring truly comparable results.

The Institute now has a number of other mobile robots that can be controlled by web interfaces. These are part of the RobOnWeb project. The RobOnWeb interface shown in figure 2.8 allows users to control one of 4 small watch battery powered mobile robots in a maze. The robots can be controlled by clicking on parts of the map, or by specifying a direction in which to move. The position of the robot is detected using computer vision algorithms applied to an overhead camera view. An image centred about the robot is pushed back to the user over a socket. This is shown on the left hand side of the interface. The bottom half of the interface provides a chat interface for talking to the controllers of the other robots.

The interface makes use of some Java 2 features and therefore requires the Java plugin to run. In addition, the front page states that a recent (version 4 or later) browser must be used. For the moment this requirement limits the potential audience of the system as the Java plugin is not currently available for all platforms, and for those it is available for, the download size is 8MB, which takes some time to download over a modem connection. There is also a versioning issue
18 Background and Related Work

Figure 2.8: The RobOnWeb interface. The system allows up to four operators to control mobile robots within a maze. The interface also provides a chat interface that enables users to communicate with each other.

The image and maps are updated at a few frames a second, providing a very responsive and interactive experience. This is achieved by using very low quality JPG images, and sending only changed map information to the client. The interface runs as three separate Applets, made to look part of a single application by the use of HTML tables and inline images as the borders for each Applet.

2.2.6 TeleZone

The TeleZone (Telezone) is a telerobotic art installation that is intended to create a parallel between a real and virtual world. Users submit requests to the robot by designing structures using
a wireframe modelling environment which runs as a Java Applet. These structures are then constructed by an industrial robot arm which cements 1cm rectangular elements within a 2 meter building area. The interface is shown in figure 2.9.

The interface is split into many parts with the use of frames. This is similar to the Khepera interface described in section 2.2.5. Users can choose the contents of each frame, allowing customisation of the interface. Contents of the frames can be camera images, virtual camera images, help, chat and other tools. The central frame is larger than the others and contains the construction interface. The selection of different frame contents is done using Javascript. The contents of the frames mostly use Java and Javascript to provide interaction. The site is designed for version 4 browsers and later.

The system makes heavy use of the fact that the environment is highly structured. Users are not exposed to control of the actual robot; they merely specify positions for blocks. Users cannot specify where these blocks come from; blocks are provided for the robot using a feeder. As the environment is so structured, it is possible to present all necessary user information from models, by the use of wireframe and virtual reality. Lower level commands are possible using a robot...
camera and the current state of the landscape can be explored by moving the robot around the workspace.

The research aims of TeleZone are somewhat different to those of traditional telerobotics. The project is interested in the behaviour and accomplishments of a virtual community that will create architectural structures by using the system. The community is provided with a framework of rules which mainly contain basic physical conditions. The participants are asked regularly to decide on modifications of the rules by a democratic vote. A main focus of TeleZone is to motivate users to interact with each other, and to study the resulting group dynamics.

2.2.7 Other Web Robots

There have been many other web telerobotics experiments, some of which are briefly discussed below:

Monteiro et al. (1997) teleoperated a mobile robot using a Java Applet. The server side was also written in Java. RTP using UDP was chosen over TCP/IP as it does not resend lost packets, provides time delay and jitter estimation, peer identification and packet loss estimates. The robot could operate in two modes: a low level mode that had a left, right, up and down interface, and a higher level mode where goals could be specified. The high level mode is still in development, due to the complexities of increasing the robot’s autonomy sufficiently. To combat the time delay of state updates, a predictive display of robot position is used. However, this is an aspect of the system that the researchers felt needed improving. The interface does not provide direct visual feedback. Instead a model of the workspace obtained from the robot’s sonar sensors is presented.

Belousov et al. (1999) developed a Java 3D interface to control a PUMA robot arm. The Java 3D model is used to provide a visualisation of the robot, that is updated by state information from the robot. Only the robot is visualised, not the remote environment. Visual feedback is also used, but lags the robot model, which only requires the low bandwidth state information. Java3D was chosen over VRML as it is more efficient and has less security restrictions. A simple model running on a Pentium 200 renders at 11 frames per second. To provide higher level supervisory control the robot is controlled using a robot control language (RCL), consisting of three types of commands - motion, location, and servicing. Motion commands are things such as ‘go A’ or ‘trot z 30’ - rotate by 30 degrees about the Z axis. Location commands allow set points to be defined so that they can be included in motion commands. Servicing commands allow the gripper to be opened/closed, the robot to be calibrated, and other non standard operations. The language was developed to be interpreted using the TCL-shell. TCL can be incorporated into Java Applets using
the JACL package. Control was tested using a controlled client in Milton Keynes UK, while the robot was in Moscow, Russia. The bandwidth was extremely limited (0.1 KB/s) meaning that it took 20 seconds per JPG image - hence the update via robot model was essential.

An interesting but rather different application is the University of British Columbia ACME (Pai et al., 1999; Pai, 1998) project. ACME is a telerobotic measurement facility for building ‘reality-based’ models. These models include information on such things as shape, texture, and sounds from interaction with the real object. The system makes use of probes, cameras, laser range finder and other sensors. In all, it has 15 degrees of freedom. The system can be controlled using teleprogramming in Java. The Java code ‘experiment’ can be tested locally in a Java 2D client and then downloaded to the actual system. The system makes heavy use of the teleprogramming concept to minimise bandwidth requirements.

Finally, Alex et al. (1998) demonstrated teleoperated micro-manipulation using a supervisory VRML interface. Java was used to program the VRML interface and to communicate with the micro-assembly workcell. The system makes use of visual servoing to enable the high precision positioning required for micro assembly. The operator interface presents a virtual model in which the user can select and drag objects. A live video image is also used for visual confirmation. As with many of the above examples, state information is used to provide a fast updating model view of the robot, while video provides a slower confirmation of the robot and its environment.

### 2.3 Other telerobots

Not all telerobots that operate over the Internet have a web interface. This section discusses some of these systems.

Fiorini and Oboe (1997) investigated the behaviour of the Internet with a view to continuous control of a telerobot using force feedback. Results of network performance tests between fixed nodes over the Internet suggested that the delay characteristics of a link can be modelled using the mean and variance of the round trip delay. They also identified a second problem not previously considered in traditional telerobotics; that of packet loss. A protocol that considers both these factors and provides some form of guaranteed performance is defined as a Real Time Network (RTN) protocol:
Real Time Network protocols are designed to connect clients with specific performance requirements, and to guarantee the fulfilment of those requirements. The performance is intended as desired throughput, delay and reliability.

They discuss two approaches to providing RTN services; either to use existing standards at the cost of accountability, or to emphasise guaranteed performance at the cost of compatibility. The real time protocol (RTP) with its associated real time control protocol (RTCP) is an example of the first type of protocol and is used for multimedia streaming over the Internet. Tenet is an example of the second type, and was designed on the premise that no RTN can be built over a data link which does not guarantee a maximum delivery time (IP does not guarantee a maximum delivery time). Tenet therefore only works on network architectures that provide this guarantee such as FDDI and ATM. Finally, they recommend the most viable approach for real-time control is to base development on RTP.

Brady and Tzyn-Jong (1998, 1999) developed a mixed architecture that uses both supervisory and shared continuous control over the Internet. Initial point move commands are sent from an autonomous planner (located near the robot), but once moving, the robot operates in shared continuous control. Feedback to an operator is via a predictive display produced via Deneb’s Telegrip environment. The main input device was a three degree of freedom joystick. The User Datagram Protocol (UDP) was used as the communications protocol. To account for variable time delay, a network model was used to provide an expected delay for control. The model for time delay is broken into three parts; nominal propagation delay, disturbance delay, and bandwidth delay.

Stein et al. (1995); Stein (1994); Sayers and Paul (1994); Lindsay et al. (1993) developed a supervisory telerobotic system, using a method of control they termed teleprogramming. The task was a puncture and slice task on a thermal blanket - a satellite repair mission subtask. A predicative display was used to counter time delay, and the level of robot autonomy was such that it could operate in the region of 30 to 60 seconds without intervention. Although designed for earth to orbit space teleoperation, the system was also used over the Internet. Tasks were performed with minimal visual feedback; the operator could occasionally transfer visual images from the remote site for inspection and validation. Delays were found to vary between three and fifteen seconds, with an average delay of six seconds. Teleprogramming was found to be effective in reducing bandwidth requirements and improving robustness to variable time delay.

Lloyd et al. (1997) implemented a model-based telerobotic system that enabled manipulation of blocks over the Internet. The research goal was to make robotic programming intuitive and easy, using a virtual environment whose model was automatically acquired using computer vision.

Networking protocols and issues are discussed in more detail in chapter 5.
User actions within the simulated environment could be easily tracked and interpreted. The model implemented 3D contact dynamics and user input used a 2D mouse. To move the robot, an object is manipulated to its required pose within the virtual environment. A local planner then generates a set of robot commands to execute the move. The main focus of the work was on model acquisition and robot programming, but additionally the interface and robot were connected via TCP/IP and could therefore be distributed across the Internet. This was demonstrated at a conference in 1996 between Montreal and Vancouver. The high level of programming, and local modelling lent itself well to the poor communication performance of the Internet.

2.3.1 Distributed Telerobotics

An area of robotics that is related to web telerobotics is the field of Distributed Robotics. Distributed robotics systems can consist of multiple robots, sensors, or agents that are physically separated and must communicate using a medium such as Ethernet. An example is a system in which multiple mobile robots exchange messages to play football (Kitano et al., 1997). Fixed base manipulators used in manufacturing are also often referred to as distributed robotic systems as there is often a supervisory system that controls and monitors individual robots via a communication bus.

A large body of research in distributed robotics is that of autonomous robots and software agents. These systems are known as DARS - distributed autonomous robotic systems. Most applications involve the coordination and control of multiple autonomous mobile robots (Wang, 1994; Piaggio and Zaccaria, 1998) but in some cases include manipulators as well (Giuffrida et al., 1994). As these systems are by definition autonomous they are not designed with human operators in mind. They also tend to be locally (within the same room or building) distributed systems where time delay and packet loss are not big problems.

Burchard and Feddema (1996) present a generic robot and motion control interface for distributed supervisory control of devices with up to six degrees of freedom. The system is based on the generic intelligent systems control (GISC) developed at Sandia labs and CORBA communications. The use of CORBA enables clients running on different operating systems to all control the same robot arm with minimal implementation differences.

Paolini and Vuskovic (1997) developed a distributed robotics lab and based some of their design on the GISC framework. This system was telerobotic in that the robot could be controlled from a separate workstation using a shell interface. CORBA was used as the communication infras-
Background and Related Work

Structure enabling different parts of the system to be split across different hardware and operating systems. The system was only locally distributed, within the lab environment.

Hopp et al. (1998) implemented a system for teleoperation of mobile robots over the Internet. They made extensive use of modelling prediction and virtual reality to reduce bandwidth requirements. Clients contained their own robot simulator to predict future robot states in the absence of update information from the remote scene. Results were visualised in a virtual reality environment, where different viewpoints could be selected. The system supported the synchronisation of viewpoints for distributed observers. It could communicate over a number of different transport protocols. The virtual environment was modelled a priori and is only partially correct as it is not updated with information from the robot’s sensors. For example, if the robot passes a door and measures the opening angle, this should be updated in the virtual environment.

The Universities Space Automation/Robotics Consortium (USARC) has developed a distributed data sharing environment for telerobotics - the TeleRobotics Interconnection Protocol (TelRIP) (Ciscon et al., 1994; Kondraske et al., 1993). The architecture has been applied to a telerobotics testbed consisting of a number of universities in the US connected via the Internet. The setup includes robots in Austin, A&M, and Rice University. Using the TelRIP protocol these robots can all be controlled from different sites; configuration being determined at run time.

TelRIP is data centric, in that data defines the interprocess connections. The key element is undirected messages: instead of the producer of data directing its messages to a list of consumers, it informs the system of the data’s presence. Data is received when clients issue requests for certain types of data. The TelRIP system handles the routing and delivery of data between different parts of the system. The atomic unit of communication between processes is a data object. The data portion can consist of many types of data, identified by a numeric code. Data types are assumed to be well known throughout the system. Name – value pairs can be added to the data object. Some are automatically created, such as a timestamp for when the object was created and a source address identifying the module that created it. The routing between parts of the system is dynamic, with each separate node running a ‘router’ process. How these nodes are connected is determined at run time, but where possible an \( n \) to \( n \) mapping is attempted. Due to performance problems across WAN networks the router network has an additional layer that handles connections with remote sites. This layer is known as Site Interface Processors(SIPs). To restrict data to parts of the system, the concepts of maps and domains are used. Maps are similar to geographic maps, and contain all processes operating under TelRIP. Each map is subdivided into domains which provide mechanisms for grouping processes hierarchically. This grouping has the structure of a tree. The system has been applied to a hierarchical path planner for a mobile robot, in which the path planning algorithm was split into layers, each layer operating on a different processor.
2.4 Summary

These different systems show a variation in both the tasks web telerobots perform and the way that they are implemented. However, their requirements of the underlying web technology are often quite similar. Despite this in most cases the web side of the system is implemented from scratch. This shows the need for a reusable framework.
The first of the two telerobot architectures to be implemented was a CGI based system. This was an evolution of the original system built by Ken Taylor (Taylor, 1999), ported and restructured to work under Windows 3.11. The system changed considerably over a number of years with new ideas and technologies. This chapter presents the details of the CGI process and how it was harnessed to control a robot. Design changes that were made to improve the system are also presented. It also discusses the user interface and the web techniques required to turn a web browser into a telerobotic control interface.

3.1 Extending HTTP and HTML

HTTP has proved incredibly flexible for different applications and it is now used in ways never originally conceived of. However it was originally intended as a way of exchanging hypertext between clients and servers over the Internet (Hyper Text Transfer Protocol). In its simplest form, a client sends an HTTP request to the webserver, which maps to a file on the server’s disk. The server reads the file, and sends it back as the body of the HTTP reply. The browser receives the file, and displays it in the browser window. This is then the end of the conversation and both client and server go their separate ways. If the displayed page contains links, and the user chooses to follow one, a new request is formed, and the process is repeated.

To use HTTP/HTML to control a robot using a web browser requires the use of a number of techniques that extend or even bypass the above client/server interaction. These can be on both the client and server side.
3.2 CGI

CGI was the original method by which dynamic content was created on the server side. CGI processes are launched by a web server when an HTTP POST or HTTP GET request are received. When a user fills in a form on a web page and then submits the form, the browser reads the values of fields within the form and includes them as name value pairs with the HTTP request. In a GET request the values are included after the URL and have a limit of 256 characters. For a POST request the values are included in the body of the request. Examples of GET and POST requests sent by a netscape browser are shown in figures 3.1 and 3.2. As can be seen, there is little difference between the two request types, other than the place that the name value pairs are included. However, the 256 byte limit of HTTP GET does mean that applications sending larger amounts of data must use HTTP POST. Historically a GET request was supposed to be for a request that did not change anything on the server side, while a POST was seen as ‘posting’ data to the server which might then be used to update something on the server side.

```
GET /cgi-win/telerobt.exe?OperatorId=OperatorIdVal&
SessionId=19991019141348&UserId=Guest&Gripper=on&
X=327&Y=169&Z=100&Spin=17&Tilt=0&Im1Zoom=0&
Im1Grey=60&Im1Size=175&Im2Zoom=0&Im2Grey=60&
Im2Size=175&Im3Grey=60&Im3Size=175&Im4Grey=60&
Im4Size=175&Interface=SingMult.htm&UserTemplate= HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.08 [en] (X11; I; Linux 2.0.34 i686)
Host: voyager.mech.uwa.edu.au:9034
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,* utf-8
```

Figure 3.1: Example GET request. Form values are included as part of the request URL. For printing purposes some lines have been split, indicated by a \.

Once a server receives an HTTP POST or GET request, if the requested file path maps to a CGI script then a CGI process is launched to handle the request. The script is launched with environment variables relating to the request. If the request is a GET then the form variables will be contained in the QUERY_STRING environment variable. If the request is a POST then they are made available via standard in. The CGI process writes its output to standard out.

The result of a CGI process consists of a header block terminated by a blank line, followed by the content. The headers are HTTP style headers and are parsed by the server before being included with other headers such as the status, content-length, date, etc. The response of a CGI script as seen by the client is shown in figure 3.3 on the facing page. Note that the CGI script only supplied
a CGI script can return any MIME type as its response, and can also add other headers. For example, the script may return images or plain text. Additional headers provide control over options such as caching of the page, and are also the mechanism by which cookies are set. Some of the headers that can be set in a CGI script are shown in Table 3.1 on the next page. Of these Content-type and Location are mutually exclusive and determine the main type of response. A Content-type header indicates that some form of MIME document is to follow, whereas a location header redirects the browser to a new location. The setting of cookies is explained in more detail in section 3.2.3 on page 32.
The CGI System

3.2.1 WIN CGI

The telerobot code uses a small extension to CGI known as WIN-CGI, created largely due to the poor performance of the DOS shell of Windows 3.11 and Windows 95. Instead of reading parameters via standard input, the process is launched with a reference to a Windows initialisation file (ini file) that contains the name value pairs. Also included is the name of a file that the process should write its output to. A typical ini file is shown in figure 3.4 on the next page. To obtain the form values, standard Windows ini file functions can be used. A disadvantage of this system is that the final output is only written once complete, so it is not possible to update pages in an incremental fashion. Also, as the mechanism for reading variables is different, some public domain CGI libraries cannot be used.

3.2.2 Generating Data for CGI

As mentioned before, each CGI process is launched in response to a POST, or GET request. As shown in figures 3.1 on page 28 and 3.2 on the page before a request is simply a set headers and in the case of POST a contained message body. A request can therefore be generated by any program that opens a connection to the server and sends POST or CGI headers. The most common technique is to use a web browser and an HTML form. Browsers process the entries in the form, URL encode them and then send them as a CGI request.

Table 3.1: Headers that can be included as the first part of a CGI script output. Content-type and location are mutually exclusive.

<table>
<thead>
<tr>
<th>Header</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-type</td>
<td>MIME type</td>
</tr>
<tr>
<td>Location</td>
<td>URL</td>
</tr>
<tr>
<td>Status</td>
<td>HTTP Status message</td>
</tr>
<tr>
<td>Set-Cookie</td>
<td>NAME=VALUE; expires=Wdy DD-Mon-YYYY HH:MM:SS GMT; path=PATH</td>
</tr>
<tr>
<td>Pragma</td>
<td>no-cache</td>
</tr>
<tr>
<td>Expires</td>
<td>x</td>
</tr>
<tr>
<td>Refresh</td>
<td>x; URL</td>
</tr>
</tbody>
</table>
HTML supports a variety of form elements. These include text fields, selection boxes, checkboxes, radio buttons, and images. All fields must be enclosed within an HTML form that has as one of its attributes the type of request to perform and the URL to send it to. For the form to be processed a submit button must be included. Ideally the browser sends a name value pair for each form element. However for elements such as checkboxes the name value pair is only sent if the box is checked. This makes it hard to distinguish between a request where no checkbox was included,
and one where a checkbox was present but not checked. Images behave in a similar way; only if the image is clicked is it present in the values sent with the request. Images act as both an input and submit element. Once an image is clicked the mouse location is encoded as two separate name value pairs and the form is sent. This means that it is not possible to click two images in a form, as the form is automatically sent after the first one.

### 3.2.3 State Management

State management using CGI scripts is awkward, due to the stateless request response nature of HTTP, and the lack of persistence of CGI process. Each request from a browser is serviced by a new CGI process. For a complex system a large portion of the CGI processes work may involve re-establishing the state of the previous request. The new request must be matched up with the previous request from the same user. Some piece of data such as a session identifier must persist across sequential requests from the same user known as a session. This identifier can then be used to re-establish the previous state from some form of persistent storage. Alternatively, all state information can pass between the browser and CGI script with every conversation. This is only practical where the amount of state information is small, is not shared with other users, and does not change between requests. Some of the more popular techniques for identifying sessions are as follows:

- Remote Address
- URL
- Hidden field in a HTTP form
- Cookies

Using remote address is included because it is often implemented as a first attempt at session management. However, it is not reliable, as multiple users may have the same IP address, due to proxies or shared workstations. Some people also seem to have dynamic IP addresses that change from request to request.

Passing information as part of the URL is a widely used solution that works quite reliably. This can be done using the query string, as used by search engines, or by adding extra path information after the script name. However this can cause problems as URLs become very confusing and might refer to old session identifiers.

A third technique of use when all pages are navigated using HTML forms is to use a hidden field. This will be included in the variables passed to the CGI script. This works very reliably provided
that the user submits their next request from the previously returned form. If they submit from a different page without the hidden field, then the state information is lost.

Cookies were introduced by Netscape Communications to solve the state management problem. A script can return a cookie using a ‘Set-Cookie’ header. If the browser supports cookies, then it will send the cookie in any future requests to the same CGI script. The expiration date and paths that a cookie are sent for can also be specified in the Set-Cookie header. Cookies solve the state management problem more reliably than any of the other methods. However, not all browsers support them, and some users disable them due to privacy concerns. Another small problem is that requests from any open windows on a browser will send the same cookie, so it is not possible to have multiple concurrent sessions from the same browser.

As there is no permanent connection between browser and server, the state represented in the browser can be out of synch with the state at the server. For applications where a user’s state may change between requests this can lead to inconsistencies. For a client to keep up to date it must make continuous requests to the server. This is known as polling and is widely recognised as a very inefficient solution.

### 3.2.4 Other Techniques for producing dynamic content

There are now many techniques for generating server side dynamic content. These are shown in table 3.2 on the following page. The most basic type is Server Side Include (SSI) where the server parses the file to be returned for known tags, and replaces them with values. This can be used for counters, or dates on a web page. A more powerful version of SSI is Enhanced SSI using packages such as PHP. These tools provide a rich scripting interface that is parsed and executed before the page is returned. Two other techniques that haven’t been mentioned are Server API and Servlets. A Server API technique involves hooking directly into the server process and taking over processing of some or all of the client request. This allows flexibility to do almost anything, but comes at the cost of complexity and lack of portability. Finally, Servlets are a Java based extension of CGI that are handled by a persistent Java virtual machine. Servlets maintain the advantages of CGI, but additionally some of the state tracking issues mentioned above are handled automatically by the Servlet engine.

The choice of which technique to use varies from application to application, and is a tradeoff between complexity, portability, power and development time. Other factors such as available software and established codebase also obviously influence the decision as well. For an applica-
### Table 3.2: Types of server side extension for generating dynamic content. Reproduced from the CGI FAQ (CGI FAQ).

<table>
<thead>
<tr>
<th>Server Method</th>
<th>Power</th>
<th>Complexity</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic SSI</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Enhanced SSI (ASP,PHP)</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>CGI</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Enhanced CGI</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Servlets</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Server API</td>
<td>v.High</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>

tion such as controlling a robot over the Internet, something which tests the boundaries of web Technology, a high power solution is definitely needed.

### 3.2.5 Frames

HTML frames allow a number of HTML pages to be displayed in separate parts of a browser window by splitting the window into frames. A top level HTML page declares a frameset of pages to view in the browser window. The frameset can also specify the widths or heights of the included frames. This can be used to separate an interface into a number of parts. One frame can then be updated or changed without affecting the other pages. As with a lot of HTML features, many web authors used them in the wrong way, and now their use is discouraged wherever possible. However, they can be very useful in enhancing experiences with CGI. When a CGI form is submitted, the whole page that contained the form is refreshed with the result from the CGI script\(^1\). While the browser waits for the reply from the CGI script the page is frozen, or even blanked meaning the user is unable to do anything but wait. In addition, any information that stays the same between requests must be sent back each time as the whole page is refreshed. By using frames only one part of the interface needs to be frozen/updated at any one time.

Sometimes more than one frame needs to be updated as the result of a request. This is not possible using basic HTML and HTTP. A client side scripting language such as Javascript (see section 3.3.1 on the next page) must be used in order to communicate between separate frames.

\(^1\)provided the CGI script returns a MIME type that the browser knows how to display
3.3 Client Side Extensions

HTML is a markup language – for marking up data to be handled in different ways. It has no procedural element to it. There are some predefined behaviours for certain HTML elements, such as links (that are followed), buttons (that can be pressed), and forms (that can be filled in). However, there is no way of redefining these behaviours or introducing new ones within pure HTML. Also, there is some control over how elements are displayed such as size and colour, but the majority of formatting is determined by the browser. To gain more control over the interface, a number of client side extensions can be used:

- Scripting – Javascript or VBscript
- Plugins – shockwave
- Programming Language – Java or COM

3.3.1 Scripting languages

Scripting languages such as Javascript are embedded within HTML and then interpreted by the browser. This is most useful for handling page events. Events can be user generated, ie when a button is pressed, or state generated, for example when the page is fully loaded. These scripting languages are becoming more powerful with each new generation of browsers. Scripts can process events and write output to the browser, but actual display of content is still left to the browser as with plain HTML. Scripting is useful for simple user interaction and provides some useful techniques for enhancing an HTML based interface. However, they do not scale well to more complex interfaces, as they are hard to maintain because there are many interdependencies which cannot be checked easily. In addition, implementation carries across different browsers and scripts that work well on one operating system and browser but will fail on another.

3.3.2 Plugins

Plugins provide a way of extending a browser for a particular MIME type, by registering as the content handler. Plugins are written using an API for a particular browser, and have complete control over display and event handling. However, different browsers have different APIs and plugins must also be ported to every browser platform. Additionally, plugins must be downloaded and installed by the user, which is often a significant time investment on the user’s part. Plugins are therefore a last resort solution.
3.3.3 Executables

Browsers now have the ability to run small downloadable executables embedded within web pages. Java Applets are the most well known of these, but Microsoft is also pushing ActiveX based executables as an alternative. Java provides a more complete cross platform solution than ActiveX which is still mostly a Windows only technology. Java also has a well developed security model that restricts Applets to certain operations. Applets provide far more control than the relatively limited scripting languages and are close to the fully functional plugin interfaces. However, they operate within the confines of a browser and do not require any additional software to be installed. This does come at a cost of speed (Java byte code must be interpreted as it is platform independent), and reliability (not all Java implementations behave identically).

3.4 Using CGI to control a Robot

The basic technique of controlling a robot using CGI, is to present the user with an HTML form with input elements for specifying robot requests. These might be text boxes, each one representing a different degree of freedom such as x,y, and z, text areas for typing higher level robot commands, or calibrated images of the workspace. The user then presses a submit button and the form values are sent to a webserver, that launches a CGI script. The CGI script reads in the request, moves the robot, and returns the result of the move and position of the robot as a new HTML page. If these results are included with a new HTML form then the user can repeat the process ad infinitum to send a series of requests to the robot.

To do anything useful with the robot some form of workspace feedback must be provided, the most obvious being visual data in the form of camera images. Thus an HTML based robot interface most likely consists of robot controls as html form elements, and camera images. This is indeed the case for most web telerobots as discussed in chapter 2. Some interfaces use graphical representations of controls instead of the basic HTML elements. This has the benefit of providing a visually intuitive interface, but means that only one control can be changed per request due to the behaviour of image input elements. To combat this, interfaces that make use of graphical controls often split the page into separate parts with the use of frames.
3.5 The UWA System

The original CGI telerobot system had been developed by Taylor (Taylor and Trevelyan, 1995; Taylor, 1999) from 1994-1996. The system ran under the Windows 3.11 using one of the first Windows based webservers (now O’Reilly website). The CGI script was a DOS executable that was in the process of being converted to run under Windows 3.11. The entire system had remarkably modest requirements and ran on a single 66MHz processor with 16MBytes of memory - apart from low level robot movement that was performed by the robot controller. Communication with the robot was via a SLIP link between the PC and controller. The code was written in structured C and ran as a single CGI executable. Some of the jobs the CGI executable performed included image grabbing, resizing and robot communication. The overall system architecture is shown in figure 3.5.

![System Architecture Diagram](image)

**Figure 3.5:** The original DOS based system as implemented by Taylor. The CGI script parses user requests, controls the robot, takes images, and returns the result to the user.

As all code was contained within one executable, this was fairly large and was launched for every CGI request to the robot. Even if a user was not controlling the robot, all the robot communication software and image grabbing software had to be loaded. Some of this software required extensive initialisation and hence added processing time to each request. This was also the reason that the CGI executable was being ported to Windows. The original robot had used a DOS based library to communicate with the robot controller. However IRB 1400 ABB robot could be controlled using a Windows DLL product called RobComm. As all the robot control code was part of the CGI executable, a decision was made to port the whole script to Windows 3.11 to make use of the new library.

The system was also fairly unstable, with frequent crashes that could be attributed to operating system instability as well as errors within the CGI code. An external watchdog timer was required to reboot the entire system on regular occasions. The monolithic structure of the code made
debugging hard as there were many interrelated parts to the system. Reliability proved to be a very important issue throughout the project.

When the telerobot system was first conceived in 1994 CGI was the only method available to generate dynamic content. In moving from the Taylor96 system to a Windows version, the HTTP/CGI combination was kept due to the significant CGI codebase that had already been developed. Increasingly, there are now alternative technologies that would be more appropriate if a system was being built from scratch as discussed earlier in section 3.2.4 on page 33.

Completion of the porting process to Windows 3.11 was only the beginning of a significant architectural and interface change to the system. Later, the system was ported again to run under Windows NT, and large parts of the codebase were restructured to be object oriented in C++. Two classes of changes were made: architectural, and functional. Architectural changes were introduced to increase flexibility, speed reliability and portability and to ease maintenance. However, they had little affect on the user robot interaction and were not directly visible to the remote user. Functionality changes affected the actual process of controlling the robot. For example, changes were made to the user interface or the types of commands that the robot understood. These changes were made over a couple of years finishing in March 1998 when design of the alternative distributed systems of chapters 6 to 8 began.

3.5.1 Architectural Changes

The Taylor96 system shown in figure 3.5 on the preceding page used CGI GET which has a number of limitations as mentioned previously in section 3.2 on page 28. To convert to using POST meant a relatively small change in how and where the CGI variables were read in. The GET request was still supported and was used for a number of special purposes, such as a URL that guaranteed to observe the robot only, and a (secret!) URL for stealing control from the current user.

As a CGI script is launched for every request, the initialising of both the robot library and image library within the script was very inefficient. They were therefore split into separate permanently running executables (servers) that the main CGI script communicated with via, first Microsoft Windows messages, and finally TCP/IP sockets. This increased the speed of processing, as initialisation only happens once, and enabled multiple applications to communicate with either the robot or image hardware. Figure 3.6 on the facing page shows the layout of the system with an image and robot server.
3.5 The UWA System

Figure 3.6: The Telerobot System after preliminary changes. The webserver launches the CGI script which in turn communicates with the robot and image servers.

Windows messaging provided a simple solution to splitting the script into transient and persistent parts, but it still restricted all parts of the system to running on the same host, and was not a portable solution for other operating systems. A later version used TCP/IP sockets as the interprocess communication method and enabled distribution across multiple machines and operating systems. Different environments and machines could then be chosen for different parts of the system. For instance, the main web server could run on a stable operating system such as UNIX, whereas image grabbing and robot control might best be performed in a Windows environment where hardware support is much better. Splitting the system into separate parts and using a cross platform communication method allows such design decisions to be made.

The use of sockets was further reinforced when the new version of RobComm was released. It was designed for Windows NT and therefore the robot server needed to run under Windows NT to use it. Initially the robot server was ported to NT and moved to a different system while the rest of the system continued to run on Windows 3.11. However, Windows NT was significantly more stable so once suitable hardware had been purchased the whole system was moved across as soon as possible. Once the system was running under Windows NT crashes disappeared, and the external watchdog became superfluous.

HTML pages were originally generated directly from printf statements within the C code. To change the layout of a returned page meant editing these statements in the source code and recompiling. Not only was this cumbersome and slow, but to test the resulting page required compiling, installing the script, and then visiting the telerobot web page to see the new layout. To speed up the process and allow multiple types of interfaces, the interface specification was separated from the compiled code using a template system. This had a number of benefits - interfaces can be redesigned without recompilation, extra interfaces can be added, templates can be viewed in a web browser as static pages, and most importantly, it is possible for users to contribute their own interfaces. The templates contain variable names that are replaced by the CGI script at
runtime. This not only allowed users to choose different interfaces, but also to design their own, requiring only a text or HTML editor. Any text can be included in the template so surprising flexibility is achieved.

These changes mostly had an impact behind the scenes, and resulted in a system that was more stable, flexible, and easier to maintain. The external watchdog was no longer required, parts of the system could be moved to different machines, and problems could be quickly traced to a specific part of the system. It was also easy to extend, and was used in the LabCam project which allowed users to move a Lego based camera using a web page. This project used the same image server process as the telerobot system, so no extra image grabbing code was required. It also used the HTML template library to generate pages, meaning the only new code required was that specific to controlling a Lego device. The same architecture was adopted, with a Lego server handling the permanent connection to the Lego interface box. Another example was the use of an Java Applet to control the robot. The Applet sent and received data from the robot using the same CGI process, but used a custom interface template that just contained the essential robot and image values, and no extra text or formatting.

3.5.2 Functional Changes

The Taylor96 system’s interface shown in figure 3.7 on the next page had two images of the robot, a wireframe representation of the robot, and a set of form boxes for x,y,z and roll,pitch,yaw to specify position for the robot to move to. Image size and quality could be changed by selecting a change image option, and going to a second page.

To make the robot interface easier to understand, and faster to use a number of changes were made. As the interface specification was now split from the actual executable, it was easy to set up new interfaces. To try a new interface, all that was required was to edit a template file in the correct directory of the webserver. The use of interfaces was extended further to allow users to select different ones, or even to design and submit their own.

The specification of orientation was originally roll, pitch and yaw - three degrees of freedom. However, for the task of stacking blocks only two degrees of freedom were necessary. These were termed spin and tilt and are explained in more detail in section 3.8.1. This reduced the complexity of the interface while still allowing the user to perform all useful manipulations. However, the reason this was possible was due to the task and in other situations such a simplification may not be possible.
Figure 3.7: An early interface to the Taylor96 system, operators were presented with two images of the robot workspace, and a wireframe model. Moves were made by filling in form fields for the next robot pose. The author was unable to find a better quality image of this early interface.
The cameras used to take images of the robot and workspace were calibrated, meaning that points in 3D space could be mapped back to image coordinates (Jain et al., 1995). This had been implemented so that users could specify robot moves by clicking on images in the interface. It also meant that the position of the robot gripper in the current image was known, but this had not been utilised. This ability gave the user more control over the image. By specifying a software zoom factor users could get images centred on the gripper, at much higher resolution, but at the same overall images size, as shown in figure 3.8 on the facing page. As this zoom was a software zoom it only had effect for images that were smaller than the framegrabbers source rectangle. Image scale ($S$) and zoom ($Z$) are related by the following equation:

\[
S = \frac{D_i}{(1 - Z)D_s + Z \times D_i}
\]  

(3.1)

Where $D_i$ is the size of the final images and $D_s$ is the size of the framegrabber image, zoom varies between between 0 and 1. For each image the user may specify final size, and zoom. Using these values, and robot position, the image server can then select the appropriate source rectangle from the frame grabber and scale it according to the above equation. This provides an intuitive way for users to specify resolution and image size. The idea was to have an interface that was independent of the physical setup, and to use terms normally associated with taking photographs (hence no mention of scale). If the camera setup were to change to include an optical zoom use of a zoom between 0 and 1 would still be appropriate.

Initially, all users of the robot were known only by their remote computer address, and had the same access privileges. To gain some knowledge about users of the robot, a registration and login scheme was implemented. This consisted of a web based registration questionnaire that was linked to a CGI script that added the user to a database and emailed them a password. Registered users then had higher access to the robot that allowed them to gain control from lower level users. This ability to have many different user levels could have been further exploited to reward frequent or creative users. A number of schools used the robot as a teaching aid, and to ensure that they had control of the robot had a higher level than other users. Local lab users obviously had the highest level and were able to control the robot at will.

Using the original form to specify a robot pose to move to, was intuitive but slow for more experienced users. This was because only one move could be specified at a time, the next move could only be typed in once the previous page had been returned. To allow experienced users to specify more than one command, a simple command script language was added to the interface to provide an alternative way to moving the robot. This is discussed in more detail in section 3.8.3 on page 50.
3.5 The UWA System

Figure 3.8: How different zoom settings affect the final image. All images are taken with a centre as close to the gripper endpoint as possible. A zoom of 0 produces an image that shows almost the whole buffer, whereas a zoom of 1 shows an image at frame buffer resolution about the gripper. Zoom has a more noticeable affect for smaller final images, and is most useful over a low bandwidth connection where large images are too slow to load.

To improve operator feedback the number of cameras was increased to four to provide alternative views of the workspace. One camera was mounted on the 3rd joint of the robot arm to follow the movement of the gripper more closely. Another area identified as important for efficient operation is error reporting for operator diagnosis (Sayers and Paul, 1994; Stein et al., 1994). Errors can occur at all levels, from a failure to communicate with a server or an incorrect command syntax, down to a joint torque overload on a joint. Some of these errors may be more important than others, and some may just be warnings. A balance must be struck between providing the user with the necessary information to diagnose the problem, and supplying larger amounts of spurious data that just cause confusion. To try and strike a balance, all errors during the execution of a request were logged to a central location. After the completion of the move only the most serious and most recent error was passed back to the operator.
3.6 CGI Implementation

Figure 3.9 shows the overall picture of how the Perth telerobot is controlled by CGI. As mentioned in section 3.5.1 robot control and image grabbing are performed by permanently running servers.

The CGI script is the central process for the whole system, a separate script being launched for each user request. Using the submitted data the script must work out who the user is, what they want to do, and if they are allowed to do it. The request must then be processed, and the outcome returned to the user. If the system is freely available on the Internet then multiple requests from

![Diagram of the telerobot system showing camera and robot setup along with communication links to the Internet. Robot control and image grabbing are both performed by permanently running servers.](image)
3.7 User State

multiple users are likely to arrive close together. In this case multiple copies of the CGI process will be running at any one time. As each request is not independent (there is only one robot, and only one person can access it at a time) these processes need some way of sharing and updating state concurrently. If no processes are running, and a new request is received then the state must persist from the previous requests. So some form of persistent state storage that can be accessed by multiple processes concurrently is needed. This could be shared memory, a shared file, a database, or even a separate server process. It was implemented as a shared status file, access to which was guarded by global mutexes ensuring that no two processes could write to it at the same time.

The state information that needed to be kept included the current operator details, robot pose, and image details. An example status file is shown in figure 3.10. The file is in the Windows initialisation (ini) format, for which there are standard API calls to read and write entries. The file is in a text format meaning that it can be manually edited if required.

```
[Operator]
UserId=jojoboss
Name=Jonathon Schemoul
UserLevel=Registered
SessionId=2000210002455
Interface=SingMult.htm
UserTemplate=
StartTime=950113495
LastTime=950114572
MoveCount=5
HostName=203.146.64.164
LastIP=203.146.64.164
Email=jonathan.schemoul@lemel.fr

[Robot]
X=100
Y=250
Z=300
Roll=0
Pitch=0
Yaw=0
Spin=0
Tilt=0
Gripper=on

[Images]
Im1File=images\img11189.jpg
Im1Size=150
Im1Grey=32
Im1DontRefresh=
Im1.x=-1
Im1.y=-1
Im1.xOffset=68
Im1.yOffset=0

[Status]
RobotON=1

[Sequence Number]
File Number=11193

;.... repeated for each camera
```

Figure 3.10: The global status file shared by all CGI processes. The file contains the current operator details, last images taken, last robot pose, and whether the robot is online or not.

3.7 User State

When the CGI script is started up it reads the operator part of the status file to determine the current operator, and compares this to the operator values that have been submitted with the CGI request
(see figure 3.2 on page 29 for an example of the types of values that are passed with a request). Based on these two sets of values the script decides who the request is from, what their status is, and what they are trying to do. Users can either be operators (submitted details match current details), new operators (the robot is available), or observers (robot not available).

The state of a CGI system is distributed and out of synch, as each client browser has its own cache of state from when the last request was made. As HTTP is a request/response protocol this state is not updated until another request is made. This can lead to situations where a number of users are looking at pages saying they are operators of the robot, when in fact someone else is now in control. Similarly, users can leave a page or go back to a previous page without the server knowing. This can lead to some confusion for users unless both their previous state and the systems current state are taken into account when deciding what feedback to give them.

Figure 3.11: State transition diagram for the possible pages a user receives. State management is performed by the both the browser and the CGI script. If either loses track of the users details then the session must be started again.

Figure 3.11 shows a state transition diagram for the possible pages a user receives and how they progress from one to another. Decisions as to what state a user should be in is decided by user priority, whether the user wants to control the robot, and idle time of the current user. Users have
a priority level, which consists of four levels: guest, registered, teaching, and team. A user can use the robot if their priority level is higher than that of the current user or if the current operator has been idle for 3 minutes.

3.7.1 Users

Once the new state of the user has been determined then the script can start processing the user’s request. Depending on their status, different operations are performed as shown in figure 3.12 on the following page.

If the user was not previously in control of the robot, wants to control robot, and the robot is available then the user becomes a new operator. Firstly, a new session is set up for the user, and the details written to the operator section of the status file. New pictures of the robot are taken and the position of the robot updated. This information is sent back to the user via the new operator page.

If the user is already in control of the robot then they are the operator. They may move the robot, change the image settings, or release control of the robot. Moves can be made by typing in coordinates for the robot to move to, specifying a set of moves using a shorthand script as discussed in section 3.8, or by clicking on points in the images. Image settings such as size, zoom and quality can also be changed.

If the user has chosen not to control the robot, or is unable to, then they are an observer. Observers can only watch what the current operator is doing, so are unable to move the robot or take new images. They receive a page containing the robot position, images, and details of the current operator. Further requests update this data if a move has been made in the meantime.

Once a user specifies that they have finished with the system they get sent a feedback page where they can leave comments about the robot, go back to the homepage, or become an observer to watch other operators.

3.8 Move Commands

Move commands are either a single position request or a series of moves (robot path). Single moves are specified as a set of coordinates, whereas multiple moves are specified using a script
Figure 3.12: Activity Diagram for CGI script. The script first establishes the state of the system and the remote user. Depending on their new state, different actions are performed before creating a new HTML page and exiting.

language. Once the robot has finished moving to the required position(s), the final position of the robot is recorded along with any errors that may have occurred during execution. Robot moves are made by connecting to a port on the socket server and sending the commands. Results are received over the same socket. Details of the communication protocol and the server architecture are discussed later in section 3.11 on page 55.
3.8 Move Commands

3.8.1 Single Moves

Single moves consist of 6 parameters: X,Y,Z, spin, tilt, and gripper position. X,Y and Z are measured from one corner of the table (XY in the plane and Z vertical). Spin is defined as rotation about the table’s Z axis and tilt is the angle the gripper makes with the Z-axis. Specifying orientation with spin and tilt ensures that a line drawn through the gripper endpoints is always in a parallel plane to the XY plane of the table as shown in figure 3.13. This is easier to understand than roll, pitch and yaw and simplifies the use of the robot. It does lose one degree of freedom, but this is not necessary for the block manipulation task. Gripper position is either open or closed.

<table>
<thead>
<tr>
<th>Spin</th>
<th>Tilt $-45^\circ$</th>
<th>Tilt $45^\circ$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-90°</td>
<td><img src="image1.png" alt="Image 1" /></td>
<td><img src="image2.png" alt="Image 2" /></td>
</tr>
<tr>
<td>0°</td>
<td><img src="image3.png" alt="Image 3" /></td>
<td><img src="image4.png" alt="Image 4" /></td>
</tr>
</tbody>
</table>

Figure 3.13: Spin and tilt orientations for the same coordinate position. Note that the jaws of the gripper are in a parallel plane to the table at all times.

3.8.2 Image Defined Moves

Moving the robot using coordinates requires users to be able to calculate these from the workspace images. To assist them a 100mm spaced grid was drawn on the table, and the sizes of blocks were given. This is fine for a lab demonstration, but is not likely to be feasible in a real world application. Firstly it is unlikely that the remote environment can be artificially marked out, and secondly the size and shape of objects is likely to be unknown. By calibrating cameras in the remote environment, features in the image plane can be mapped to features in the environment, for example a point in an image maps to a line in the workspace. A workspace point identified in two images, maps to two lines in the workspace. The workspace point can be estimated from
where these two lines meet, or in reality the point where they are closest together. This is referred to as the stereo point algorithm (Appendix B has details of its implementation).

This was used within the HTML interface by making the images part of the form. If an image is clicked the mouse coordinates are sent as part of the request. An image form element causes an immediate submit, so as stated previously, it is not possible to click two images and then submit the form. As specifying a real world point requires two separate images, the user must be sent a second page to specify the second point. The first point is remembered via the status file, and once the second point has been sent via the second page, the 3D point is estimated and a move request sent to the robot, as in section 3.8.1.

This system is rather convoluted and slow as the user must wait for two pages before submitting the move request. Also, continuity was lost between the two page updates as users often forgot where they had clicked in the first image by the time the second page was received. A further problem was that as soon as an image was clicked and the request was submitted, there was no way to check that the right part had been clicked, or to change its location before submitting. Using Javascript and HTML layers an interface was built that handled the image clicks locally. A cross was displayed on the image using HTML layers so that users could see where they had clicked; both images could be marked before the request was submitted. Unfortunately, due to differences in how Javascript and layers was implemented across different browsers, a solution that worked for all browsers proved difficult to find.

### 3.8.3 Multiple Moves

The syntax for the multiple move script is as follows:

```
<command><newvalue>;<command><newvalue>
```

or

```
<command><newvalue>\ superior<command><newvalue>
```

Each move is separated by a semicolon (or by spaces\(^2\)). Any robot parameters omitted are kept the same as the previous move. For example: if the current position is \((x=0 y=0 z=0 \text{ spin}=0 \text{ tilt}=0)\) then \((y100,z100)\) translates to a request of \((x=0 y=100 z=100 \text{ spin}=0 \text{ tilt}=0)\). Possible parameters and values are shown in figure 3.14 on the next page. A more complex example of a script might be something like:

\(^2\) superior denotes a space
This is a sequence for picking up a block and standing it on its end. It assumes that the gripper is aligned and over the top of a block 20mm from one end. It contains six separate moves that grip the block with a tilt of 45 degrees, move up to a safe height of 100mm, rotate the block through 90 degrees, place it back and finally open the gripper.

<table>
<thead>
<tr>
<th>Action</th>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>open gripper</td>
<td>g</td>
<td>0</td>
</tr>
<tr>
<td>close gripper</td>
<td>g</td>
<td>1</td>
</tr>
<tr>
<td>open gripper</td>
<td>g</td>
<td>open</td>
</tr>
<tr>
<td>close gripper</td>
<td>g</td>
<td>close</td>
</tr>
<tr>
<td>move x</td>
<td>x</td>
<td>-50 to 550</td>
</tr>
<tr>
<td>move y</td>
<td>y</td>
<td>-50 to 550</td>
</tr>
<tr>
<td>move z</td>
<td>z</td>
<td>0 to 450</td>
</tr>
<tr>
<td>move spin</td>
<td>s</td>
<td>-89 to 89</td>
</tr>
<tr>
<td>move tilt</td>
<td>t</td>
<td>-45 to 45</td>
</tr>
<tr>
<td>tilt +45</td>
<td>t</td>
<td>back</td>
</tr>
<tr>
<td>tilt vertical</td>
<td>t</td>
<td>vertical</td>
</tr>
<tr>
<td>tilt -45</td>
<td>t</td>
<td>forward</td>
</tr>
<tr>
<td>relative</td>
<td>r</td>
<td>—</td>
</tr>
</tbody>
</table>

**Figure 3.14:** Syntax for individual commands used to program multiple moves.

### 3.9 Images

After the robot has been moved, images from around the workspace are recorded to provide the operator with feedback on the result of the move. Image size and quality can be specified for each camera by the operator. Individual cameras can also be turned off, to reduce bandwidth and improve response time. The required settings are included with the CGI request variables. The process of taking images is handed to a server process that runs all the time, and is shared with other applications such as the LabCam.

Some of the cameras are calibrated. For these cameras, the images can be optionally zoomed in on the gripper end point as discussed earlier in section 3.5.2. For these images to be used in conjunction with the stereo point algorithm, their offset and scale needed to be remembered, so that the camera matrix could be adjusted to the correct coordinate mapping.
Observers only receive the last images taken - this reduces load on the server by only taking images when the robot is moved. Filenames, sizes and other properties of the images are maintained via the status file. Browsers tend to cache images for efficiency, so if an image filename is returned that matched one already in the browser’s cache then the image from the cache will be used. For images of the robot it is essential that images are not cached. Initially, a random filename was generated, which worked for users who did not make many moves, but over a longer period of time it is not guaranteed to produce filenames that have not already been used. A more reliable solution is to sequence filenames, and choose a suitably long period before repetition. This requires that the last sequence number is persistent between script invocations: another use for the status file!

Images are returned via an HTML page as image tags. The tag includes the URL of the the images to load, so once the browser receives the HTML text, it then makes further requests for each referenced image. This means that a user does not receive any results until the robot has been moved and images taken. Taking images was rather a slow process as all cameras used the same framegrabber. Time was required for the board to regain sync between each image, meaning that a second or more was required for each camera. To return a page to the operator earlier, the image URL within the page can instead refer to another CGI script that will actually take the image. The page can then be returned immediately after the robot has been moved. The script for taking images uses CGI GET to determine which camera to use and information such as size and quality are read from the status file as before. The image script returns an image mime type header along with a no-cache header, and writes new details to the status file. A comparison of the techniques is shown in the timing diagrams of figures 3.15 and 3.16. Unfortunately, this technique only works for some browsers, as sometimes the results of the image are cached, despite the no-cache header returned with the image. To allow for this, the main CGI script tests for the type of browser, and then decides whether to take the images synchronously or asynchronously.

3.10 HTML page

The final job of the CGI script is to write the reply to the client. This is an HTML page that the user’s browser then renders and displays. The page contains all the updated state information after the move has been made, including the robot pose, current images, and details of the current session. Depending on the status of the user, the type of page they receive will differ.

These different pages are created with the use of templates. The basic principle of the template is that certain parts of the template are substituted for values of variables in the CGI script. This is the return path by which state is passed back to the user. The passing of these variables continues
3.10 HTML page

Figure 3.15: Timing diagram for the complete request being processed in one single CGI script. The result is not returned until the robot has been moved and all images taken.

Figure 3.16: Timing diagram when the CGI request is chained into a series of requests. The first CGI script returns a page that contains references to a second CGI script that takes the images. The main HTML page can be returned as soon as the robot has stopped moving.

backwards and forwards every time the user makes a request. For the browser to pass variables back again they must be part of the HTML form that is submitted\(^3\). The form must therefore contain entries for all variables that need to be passed back to the script. Entries in an HTML Form can be one of a number of types, such as textfields, checkboxes, radio buttons, or hidden fields. Each entry in a form has a name, which is used as the first part of the name/value pair that is sent as part of the CGI request.

\(^3\)This is not strictly true. Cookies are also passed by the browser and are discussed in the next chapter.
Figure 3.17: Path of state information between browser and CGI script. State information is passed forwards from the browser to the CGI script via the POST process. State is returned by substituting variable values in an HTML template, which is then returned to the browser.

Figure 3.18 shows part of an operator template. This part has two purposes. When the page is first loaded by the user it shows the current pose of the robot so the user can then change any of the values to create a new pose for the robot to move to. When the template is processed by the CGI script, for each pose variable the code looks for something ending in Val and replaces it with the current value. This process is performed for robot pose, image details, and session details. Sometimes, instead of replacing a tag with the variables value, it needs to be replaced with an HTML representation of the value, such as a checkbox, or selection. These alternative replacements end in different words and are dealt with differently, as shown in figure 3.19 on the facing page.

Figure 3.18: Part of HTML Template before being processed by the CGI script. The script replaces any text ending in Val with the corresponding variable value.

Different directories are used for each type of template, i.e., there is a directory for operator, new operator, and observer templates. A given style of interface uses the same name of file in each of the directories. If a user’s control status changes, they still maintain the same style of interface, as all that changes is the source directory of the template. Different styles of interface can be chosen from a selection box that is part of the HTML page. Users can also design their own interface, and submit them via a web page. They are then immediately available to test out.
### 3.11 Robot Server

<table>
<thead>
<tr>
<th>Type</th>
<th>TagExtension</th>
<th>Arguments</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Val</td>
<td>value</td>
<td>Tag replaced by Value</td>
</tr>
<tr>
<td>Selection</td>
<td>Selection</td>
<td>Selected value and all other possible values</td>
<td>Tag is replaced by (&lt;\text{OPTION}&gt;) value pairs, with an (&lt;\text{OPTION SELECTED}&gt;) for the selected value</td>
</tr>
<tr>
<td>Checkbox</td>
<td>Check</td>
<td>Boolean</td>
<td>if checked tag is replaced with (&lt;\text{CHECKED}&gt;) else it is replaced with the empty string</td>
</tr>
<tr>
<td>CheckboxVal</td>
<td>Val</td>
<td>Boolean</td>
<td>if checked tag is replaced with ‘on’, this provides hidden checkboxes ‘on’ is the value sent by a browser if a checkbox is ticked.</td>
</tr>
<tr>
<td>Time</td>
<td>CurrentTimeVal</td>
<td>none</td>
<td>Tag is replaced with current time</td>
</tr>
</tbody>
</table>

**Figure 3.19:** Different Tag types for HTML templates. The TagExtension is the string added to a given label. For example, the tag GripperVal would be replaced with the values of the gripper (0 or 1) while GripperCheck would be replaced by \(<\text{CHECKED}>\) if the gripper variable was 1 or it would be replaced by the empty string.

An example interface is shown in figure 3.21 on page 57. This was the default interface, and proved popular with users. The interface is clearly divided into functionally separate parts. The top part shows images from the workspace, the middle part contains the robot state and controls, and the bottom part contains the less often used image settings.

### 3.11 Robot Server

All robot calculations and communication with the S4 controller are carried out by the Robot Server. The CGI script communicates with the robot server via a TCP/IP socket. Connections between the Robot Server and the S4 controller go over Ethernet to an intermediate PC that maintains a SLIP connection to the S4 controller over a serial line. The Robot Server to S4 communication is managed by ABB’s Robcomm software. The flow of control messages is shown in figure 3.22.

#### 3.11.1 Control

The S4 controller operates by running RAPID programs. To move the robot a simple program is executed. The first operates the gripper by switching the value of a digital output and then moves the robot to a target pose. To move the robot in a straight line the following RAPID command is used:

```plaintext
MoveL targetpos,vslow,fine,grippertool\WObj:=table;
```
<table>
<thead>
<tr>
<th>Tag</th>
<th>Replaced by</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SessionIdVal</td>
<td>SessionId of submittor</td>
<td></td>
</tr>
<tr>
<td>SubmittedUserIdVal</td>
<td>UserId of submittor</td>
<td></td>
</tr>
<tr>
<td>SubmittedNameVal</td>
<td>Full name of submittor</td>
<td></td>
</tr>
<tr>
<td>SubmittedUserLevel</td>
<td>User level of submittor</td>
<td></td>
</tr>
<tr>
<td>SubmittedUserErrorVal</td>
<td>Validation Error</td>
<td></td>
</tr>
<tr>
<td>UserTemplateVal</td>
<td>alternative public directory template</td>
<td></td>
</tr>
<tr>
<td>InterfaceVal</td>
<td>standard template type</td>
<td></td>
</tr>
<tr>
<td>InterfaceSelection</td>
<td>standard template type as a selection</td>
<td></td>
</tr>
<tr>
<td>JustWatchVal</td>
<td>Whether user is just observing</td>
<td>'on' for true</td>
</tr>
<tr>
<td>JustWatchCheck</td>
<td>As above but replaced as checkbox value</td>
<td>CHECKED for true</td>
</tr>
<tr>
<td>ControllerUserIdVal</td>
<td>UserId of operator</td>
<td></td>
</tr>
<tr>
<td>ControllerNameVal</td>
<td>Full name of operator</td>
<td></td>
</tr>
<tr>
<td>ControllerNameVal</td>
<td>Name of Current operator</td>
<td></td>
</tr>
<tr>
<td>ControllerIDVal</td>
<td>User Id of current operator</td>
<td></td>
</tr>
<tr>
<td>MoveCountVal</td>
<td>No of moves by current operator</td>
<td></td>
</tr>
<tr>
<td>HostNameVal</td>
<td>Hostname of current operator</td>
<td></td>
</tr>
<tr>
<td>LastTimeVal</td>
<td>Time of operator’s Last Move</td>
<td></td>
</tr>
<tr>
<td>StartTimeVal</td>
<td>Start Time of operator’s session</td>
<td></td>
</tr>
<tr>
<td>CurrentTimeVal</td>
<td>Current Time</td>
<td></td>
</tr>
<tr>
<td>ErrorVal</td>
<td>Error Messages</td>
<td>Operator pages only</td>
</tr>
<tr>
<td>XVal</td>
<td>X Coordinate of the robot</td>
<td></td>
</tr>
<tr>
<td>YVal</td>
<td>Y Coordinate of the robot</td>
<td></td>
</tr>
<tr>
<td>ZVal</td>
<td>Z Coordinate of the robot</td>
<td></td>
</tr>
<tr>
<td>SpinVal</td>
<td>Spin orientation of the robot</td>
<td></td>
</tr>
<tr>
<td>TiltVal</td>
<td>Tilt orientation of the robot</td>
<td></td>
</tr>
<tr>
<td>GripperCheck</td>
<td>Checkbox value for gripper</td>
<td>CHECKED for open</td>
</tr>
<tr>
<td>GripperVal</td>
<td>POST value for griper</td>
<td>'on' for open</td>
</tr>
<tr>
<td>Im1FileVal</td>
<td>Image filename</td>
<td></td>
</tr>
<tr>
<td>Im1SizeVal</td>
<td>Image Size</td>
<td></td>
</tr>
<tr>
<td>Im1SizeSelection</td>
<td>Image Size as a Selection</td>
<td></td>
</tr>
<tr>
<td>Im1ZoomVal</td>
<td>Zoom values</td>
<td></td>
</tr>
<tr>
<td>Im1ZoomSelection</td>
<td>Zoom value as a selection</td>
<td></td>
</tr>
<tr>
<td>Im1GreyVal</td>
<td>Image Quality</td>
<td></td>
</tr>
<tr>
<td>Im1GreySelection</td>
<td>Image Quality as a selection</td>
<td></td>
</tr>
<tr>
<td>Im1Don1RefreshCheck</td>
<td>Camera off</td>
<td>CHECKED for on</td>
</tr>
</tbody>
</table>

![Figure 3.20](image.png)

**Figure 3.20:** Tags that are replaced by the current telerobot system values in HTML templates. Some tags are replaced in both HTML form and CGI POST form. This allows the option of hidden inputs for interfaces where a particular value does not need to be changed.

This command means: move to pose 'targetpos' with velocity 'vslow', with tolerance of final position being 'fine', using tool 'grippertool' and workspace 'table'. Each of these variables are RAPID structures. The 'targetpos' structure contains the final pose, and orientation of the gripper. The 'vslow' variable contains maximum linear speed, and maximum rotational speed of the tool endpoint. The 'fine' variable contains position and orientation tolerances for the final gripper position. This is most useful for multi-pose paths where the gripper is only required to move close to and not through a way point. The 'table' variable is structure containing the position and rotational offsets of the table from the robot’s base.
3.11 Robot Server

**Figure 3.21:** The default interface for controlling the robot. The interface is divided into functionally separate parts. The top third contains images from the workspace, the middle section is for robot state/control and the bottom third contains controls for changing image settings.

**Figure 3.22:** Flow of control messages from the CGI script to the robot. The robot server listens for commands from the CGI script, or other applications, and forwards them to the Robcomm control. Robcomm then communicates with the robot via the intermediary Linux machine that hosts the SLIP link.

Only the 'targetpos' variable is changed for each move. The speed is therefore fixed, and was chosen as a balance between collision forces and path execution time. For a more sophisticated application, control of the speed per move might also be important, in which case the speed variable would also be changed for each execution.

This procedural style of programming a robot makes lower level robot control hard to implement. It is extremely hard to modify the robot's execution based on information from force, or vision...
sensors. However, this is the only published way of controlling the arm using the S4 controller.
The sequence of events to move the robot from a PC using Robcomm, are as follows:

- check/change-to run mode
- change target pose and gripper variables
- run program
- wait for program to finish
- check for any errors

At the suggestion of a user, an additional step was added to a basic move. This is known as a
RAPID\(^4\) move, a term from CNC programming which splits a path into a horizontal and vertical
component. If the gripper is moving down, the horizontal path is performed first, and conversely if
the gripper is moving up, the vertical path is performed first. This reduces the chance of accidental
collisions while executing the path, as the gripper is kept as far from the table for as long as
possible.

During execution of a move there are a huge number of warnings/errors that can occur. They can
be from the request itself, from communication problems between the PC and robot controller,
or there can be problems associated with the actual physical move. Errors in requests may be
due to workspace boundaries or incorrect syntax. Typical physical problems include singularities,
joint limits, and joint torque overloads. If the error occurs during the physical move, then the S4
controller provides an error code and some explanatory text. Often this text is quite technical and
of little use to users with little or no knowledge of robotics. For common errors, this explanation
is replaced with a (hopefully) more understandable message. For each move an error log is kept.
Errors that do not stop execution of the request are logged as warnings. Most warnings are gen-
erated when the request is checked for validity, for instance if the requested pose is outside the
table workspace. At the end of a move, the log is checked and if it is not empty the most recent
error is returned to the user. This ensures that the user only has to understand one error per move,
and that this will have been the most relevant to the failure of their request. For example, if a user
requests a move to a pose below the table, the pose will be adjusted and a warning logged. If the
robot then collides with a block during execution, then the collision error will be returned to the
user, whereas if the move completes successfully, the workspace warning will be returned.

\(^4\)This is an unfortunate clash in terminology, as it is also the name of the robot programming language
3.11 Robot Server

3.11.2 Communication Protocol

The communication protocol between the Robot Server and the CGI script follows the scheme used by Internet services such as FTP and telnet. All data and commands are passed as ASCII, sacrificing speed for easy debugging and observation.

Communication with the server uses the following scheme. Client sends a command (up to 4 characters) and then waits for a reply, before sending the command arguments. All server status replies are of the form `<major code> <minor code> <text message>`. There are four supported commands:

- **MOVE**  Move the robot to a single pose
- **POS** Get the current position and orientation of the robot
- **PATH** Move the robot along a path defined by a sequence of robot poses
- **JNT** Get the current joint angles of the robot

The replies are interpreted as follows. The **major code** can be either 0, 1 or 2. 0 indicates success, 1 is a warning and 2 is an error. The **minor code** is a detailed error code specifying the exact type of a given error. The text message gives a human readable explanation of the error, or next expected data for the case where everything is okay. A typical client server conversation for a single MOVE command is as follows:

```
server: 1 0 Robot Server (version 1.0) ready
client: MOVE ;make a single move
server: 1 0 Pose to move robot to
client: 0 0 222 1 0 45 0 0 ;required pose x y z ...
server: 1 0 Move Data OK. Moving robot... ;robot is now moving
server: 1 0 Move completed ;error messages appear here
client: okay
server: 0 0 222 1 0 45 0 0
```

Each client request has a corresponding reply from the server. For the actual physical move two replies are sent by the server, an initial request received followed later by a message to indicate the move has finished including any errors that may have occurred.

Other commands use the same command response system. **PATH** commands are a little more complex with the path data being sent in two steps. Firstly, a header indicating the number of poses to expect is sent, then the actual path data follows as a list of poses. While a **PATH** command
is being executed the server sends a message after each pose has completed. This means the client is aware of exactly which part of the path is being executed.

The Robot server contains a number of threads, one to manage the listening socket (max 1 connection at a time), one to manage the user interface which includes the Robcomm ActiveX control, and a number of other threads used internally by the Robcomm control to communicate with the actual robot. Due to limitations in the Robcomm control the instance cannot be shared across threads so all calls to robot functions must be made from within the user interface thread. However, most commands are initiated in the socket thread, so some sort of intra-thread communication is required. This is achieved by sending windows messages from the socket thread to the main user interface thread. Future versions of Robcomm will allow the instance to be shared across threads so this method should no longer be necessary.

A number of signal events are used to synchronise the two threads. For instance, the server thread calls a move command in the user interface thread and then waits for a move completed signal before continuing. This ability to wait for certain events makes the control code simpler to implement, as no polling loops are required. A typical example is changing the S4 controller from standby to run mode which can typically take a second or so, the control code requests run mode, then makes a call to wait for run mode to happen, and then carries on.

### 3.12 Image Server

The Image server interfaces between a number of applications and the frame grabber hardware. The communication model is similar to the robot server with a listening socket waiting for image requests. Clients connect to the image server, wait for an acknowledgement, and then send a command and its respective parameters. They then wait for acknowledgement that the command has been completed and disconnect or send another command. All commands take a camera number as one of their arguments, as cameras are multiplexed via a video switcher into the frame grabber as shown in figure 3.23. The video switcher is controlled from the image server via a digital I/O card. The possible commands include various image grabbing functions and for calibrated cameras, the conversion of image points to world frame coordinates.

Calibration is performed by moving the robot to a number of locations in the workspace, 12 points are required to define the camera matrix but greater than 20 points are normally required to account for noise. For each robot position the camera coordinates are recorded. These points are then used in a least squares algorithm to estimate the camera matrix (Bolles et al., 1981). By using the robot
3.12 Image Server

![Image Server Diagram](image.png)

**Figure 3.23:** The image server hardware setup. Camera images are multiplexed by a video switcher controlled from a digital I/O card. Images are grabbed using the frame grabber card.

to perform the calibration, implicit linear calibration errors in the robot are also accounted for in the camera matrix.

Images can be saved in either GIF or JPG format. The format is specified by the filename extension given as a command argument. Compression level or number of grey levels can also be specified giving the user control over image quality verses size. JPEG encoding was found to offer the best compression for typical workspace images and is the current default for the CGI script.

3.12.1 Communication Protocol

The image server listens on a socket and can execute a number of commands. Commands and replies are sent using ASCII as with the robot server (section 3.11.2 on page 59). Replies are also in the same format of `<major code> <minor code> <text message>`. The set of commands is:

- **XYZ**: Take image about world point \((x, y, z)\)
- **OLD**: Previous image request format (backwards compatibility)
- **U2X**: Convert two \((u, v)\) points to a world point \((x, y, z)\)
- **STD**: Take a centred square image
- **RLY**: Switch to a new camera number
- **FUL**: Take an image from the whole area of the buffer

Table:These commands take a varying number of parameters. Image taking commands require a filename and final image size, plus some specific arguments such as the \((x, y, z)\) point and required software zoom for the XYZ command. To convert \((u, v)\) points back to a world point \((x, y, z)\) the image scale, and offset must be supplied so that the correct \((u, v)\) coordinate frame is used.
Due to synch problems with the frame grabber, camera switching needs to be performed as soon as possible after the last image is taken, thus most commands take an argument that is the next camera relay to switch to, which is performed before the last image is compressed and saved to disk.

### 3.13 Live Video

With a static HTML interface, users never actually see the robot move. They only see images of before and after a move. This reduces possible timing confusion as the images are always taken after the robot has stopped moving. However, if the movement made is small, or there something unexpected has happened it is hard to pick up from static images. In addition, users want to see that they are really moving the robot; they want live video. Live video can now be displayed within browsers in a number of different ways, using specialist plugins or even using the basic capabilities of standard browsers. Webvideo (Wolf and Froitzheim, 1997) is a free tool that has been designed specifically for remote control Internet operations. It has been designed to minimise delay, and account for different connection bandwidths, while working with standard browser technology. It uses animated GIFs and a permanent HTTP connection to each browser to push new images to the client. This was most easily integrated with the CGI script by splitting the controller page into frames. The top frame includes the standard CGI control page and the bottom frame shows the live video, while a move is being executed, live video shows the move in the bottom frame, and once completed the top frame refreshes with new images and the results of the move. This change required no alteration of the CGI script and just required a few changes to HTML pages, as well as the installation of the Webvideo server. This shows one of the advantages of a CGI, HTML interface. It is very easy to change significantly without any need to recompile, or redesign executable code.

---

5 Incidentally on a different machine to the main telerobot server
CHAPTER 4

CGI Results and Discussion

This chapter presents results and experiences of running and developing the CGI telerobot system between November 1997 and July 1998. The system was continually evolving during this time, as outlined in the previous chapter. The goals of these changes were to improve speed, reliability, and usability. The first part of this chapter presents usage results, and how they were effected by changes to the system. The second part discusses errors and problems that occurred and how these related to the architecture of the system.

4.1 Telerobot Usage

The first statistic of interest in assessing a web telerobot is how many people were able to use it. What is of interest is not so much how many people use the system, but the ratio between the people that intend to use it, and those that actually do. Figure 4.1 shows the number of hits per week on various pages relating to the CGI system. The top line shows number of hits on the front page (median 2065), the middle line is the CGI script itself (median 1191), and the bottom line shows new operator pages (median 697). From this we can see there is a drop off of \( \approx 40\% \) from the front page to the CGI script, and a further drop off \( \approx 40\% \) from the CGI script page to the new controller page. This means that about \( 1/3 \) of users who viewed the front page went on to control the robot.

Taylor (Taylor, 1999) presented in depth analysis of the results from a comparison of his Taylor96 system and this CGI system. His predominant interest was in how people interacted with a web Telerobot, and not how the different systems compared. However, he did suggest a number of techniques for evaluating a telerobot system. These included measuring the time between requests, measuring the number of moves in a session, and obtaining subjective feedback from users. These methods were easier to use than more traditional techniques such as measuring task completion.
Figure 4.1: Number of hits per week on different parts of the telerobot system.

time (Drascic et al., 1989; McLean et al., 1994), because the task undertaken by operators is not well defined, and the environment is not strictly controlled. The main technique that Taylor used was to fit a Weibull (Walpole and Myers, 1972) distribution to the number of moves in a session. He found the shape parameter to be broadly similar across different systems, but found the scale parameter varied. A smaller scale factor indicated a tendency for longer sessions, and therefore a more usable system. He found the scale parameter for ABB 1400 system was four times smaller than that of the original Taylor96 system which indicated that sessions were significantly longer with the new system. This verifies that the initial changes made to the system made a positive difference (see sections 3.5.1 on page 38 and 3.5.2 on page 40. To summarise, the changes that were made to Taylor’s original system included:

- Software camera zoom
- Interface templates
- Spin and tilt specification for orientation
- Increase in number of cameras to four
- Robot error feedback
- Separate robot server
- Separate image server
Figure 4.2 shows the moves in a session plotted for 3 six month periods once the new CGI system was in operation. During this time the system was being changed incrementally, with the major change being a port to the NT operating system, and changing the image and robot servers so that they used sockets (see the project time-line shown in figure 1.2 on page 6). Operator visible changes included the introduction of multiple moves, user accounts, and many new template based interfaces (users were encouraged to submit their own interface templates). These changes seem to have had minimal impact on the length of operator sessions as the scale parameter is similar in all three cases.

![Graph showing moves per session and Weibull distribution parameters for three periods.](image)

**Figure 4.2:** Histogram of moves in a session, plotted on a log scale. The three figures show the distribution of moves in a session for different stages in CGI system development. Also shown is the maximum likelihood estimate (MLE) fit of a Weibull distribution (along with the 95% confidence intervals).

Figure 4.3 on the following page shows histograms of the time between moves for three six month periods. This data was not available for the Taylor96 system so it is not possible to make a comparison. The median time between moves for each of the three 6 month periods is 46, 45 and 47 seconds respectively. This again shows little variation over the year and a half period. At the time it was hoped the faster hardware used for the NT version would significantly improve the time between moves. The fact that it did not suggest that request processing time was not the governing factor in the time between moves.
A statistic that was not analysed by Taylor was the distance moved by the gripper endpoint. This provides a measure of what users are doing with the robot. It also gives a better idea of the reliability of the system, as only when the robot is moving can you be sure that the system is fully operational.

Figure 4.4 shows the distance moved by the gripper estimated on a weekly basis. This was calculated by summing distance between adjacent positions recorded in the log files. This gives a conservative estimate of the distance moved, as it does not account for multiple or rapid moves. Measuring distance moved measures a combination of reliability, popularity, and ease of use of the robot. It can be viewed as an indication of overall usability of the system. The figure shows a general upwards trend in the distance moved from around 300 metres per week, to 500 metres, with a peak of nearly 600 metres. There are some periods where the distance is zero. These correspond to weeks for which no logs are available due to human error, and hard drive failures. The overall upwards trend shows that the reliability, and usability of the robot continued to improve as the system developed.
It is also interesting to look at the final positions of moves operators made, as this can be used to give an indication of what they were trying to achieve. Given that the blocks on the table are in random positions, the positions moved to ought to be random as well. Figure 4.5 on the next page shows a histogram of final move position for the $x$, $y$, $z$, spin, and tilt. These histograms show that the moves made by operators were not random at all, with strong peaks at regular intervals. These correspond to grid lines that were printed on the table, and show operators were heavily reliant on these to navigate in the workspace. The spin and tilt values also show sharp peaks, which correspond to predefined choices that were available in some of the interfaces.

The ultimate test of how usable the telerobot system was, was what users built with it. Some of the more accomplished creations are shown in figure 4.6 on page 69. These structures show what human ingenuity is able to achieve. Some of these structures would be hard to build by hand, let alone using a web browser to control a robot arm on the other side of the world! However, it should be noted that these took some time to build, (5 hours for the pyramid for example) and immense patience was required on the part of the operator.

Figure 4.4: Distance moved per week from January 1997 to April 1998. This gives a combined indication of reliability, usability and popularity of the robot.
Figure 4.5: Histogram of robot poses moved to by operators. The type of moves made are heavily influenced by visual cues in the workspace, and options available in the interface. X and Y positions show a strong correlation with the 50mm spaced grid drawn on the table.

4.2 Reliability and Errors

As alluded to in the previous section, making the system reliable proved to be a significant challenge, the goal being to have the robot available 24 hours a day, 365 days a year.
4.2 Reliability and Errors

Figure 4.6: A selection of the more accomplished creations. Some of these structures are so precarious it is hard to imagine building them by hand, let alone by using a two fingered gripper on the other side of the world, controlled using a mouse and a web browser.
Within the telerobot system there were at least 6 different software subsystems as well as 4 pieces of hardware involved in executing robot request. To function correctly these subsystems all needed to be fully operational. Observation and correction of problems can achieve local software reliability, but for third party products this is not possible. Both the web server computer, and robot operating systems generated occasional faults that required a manual reset. A hardware and software watchdog were therefore required to force a reboot if either system stopped responding. The need for a webserver watchdog disappeared once the system was moved to the NT operating system, and similarly the robot watchdog was no longer required once the robot fault had been identified and new hardware/software installed. However, there were still occasional catastrophic faults in the robot operating system, that could only be fixed by a manual reboot.

### 4.2.1 Errors

Although the system may be functioning correctly there are still a large number of errors that can occur during a move. These can range from minor errors such as a move request outside the workspace to more serious errors such as the robot reaching joint limits and singularities.

![Types of Robot Errors](image)

**Figure 4.7:** Types of error during a move. The most frequent errors are due to operators trying to move outside the restricted workspace, and collision errors. Other significant errors, are in use errors caused by requests being sent close together, robot joint limits, and safety errors caused by the light curtain being broken.
4.2 Reliability and Errors

Figure 4.7 on the preceding page shows the breakdown of the type of errors that occurred during moves. The most frequent errors were workspace limit errors, closely followed by collision errors. The third most frequent error, the in use error, is a result of a user sending the same request twice. This is easy to do with a web browser by accidently pressing the submit button twice. It causes all sorts of problems on the CGI script side as this results in two CGI scripts being launched with the same parameters. The solution was to send an error page back to the user if they submitted twice. This error page informed them that the robot was in use, probably because they sent two requests close together. This illustrates one of the many problems that were encountered trying to handle all browser requests logically.

Other errors that occurred frequently were light curtain, and joint limit. The light curtain error was caused by the safety light curtain being broken by somebody working in the robot workspace. Unfortunately they often forgot to reset it once they had finished. The joint limit error was caused by certain sequences of events that would put the robot into positions that it could only be extricated from with manual intervention. Avoiding this problem was an interactive process of limiting certain moves, and adding various reconfiguration moves to the robot command library. This was an interesting consequence of having the robot on the web and allowing anyone to control it, and is probably best summed up by the following quotation:

“A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.”

Douglas Adams, Mostly Harmless

4.2.2 Closure

Due to the variation in knowledge and motivation of Internet users, a device controlled over the web is guaranteed to get into unimaginable states. To provide a reliable service, the system must be able to recover from all these states. This applies not only to software control systems, but to the device and its environment as well. This is similar in principle to that of mathematical closure, where a given space can be spanned by a set of operations - the key is to find the set of operations that constitute a 'closed' system.

In the telerobot application, this meant the robot must recover from singularities, joint limits, and collisions. Also, for the users to be able manipulate blocks on the table, there must be blocks to manipulate. This was a constant problem throughout the project, as a large proportion of users found it easier/more entertaining to try and push all the blocks out of reach, than to build something with them.
Initially, the table was flat, with the robot restricted to a central portion of its area. There was a space of about 100mm round the edge of the table that was not accessible. Despite this, users became quite proficient in removing blocks from the table, either by pushing blocks off the edge by using several blocks in a row, or by picking up blocks and dropping them from a height. With this configuration, picking up and replacing the blocks was a daily activity, mostly in the morning after the busy overnight session when America was online. To try and keep the blocks on the table, its size was increased using a perspex sheet with the corners turned up at 45 degrees. The inaccessible border was now 200mm on each side. This reduced the frequency at which blocks needed to be picked up of the floor to once a week, but it was still relatively easy to push the blocks into the out of reach zone.

To try and close the system, there needed to be a way of getting out of reach blocks back into the correct area, without allowing a user to push the blocks even further away. One solution that was tried was to define a fixed robot path that swept blocks back into the centre of the table using a blade much like a wind-screen wiper. It was hoped that this would close the system completely eliminating the need for human intervention. However, due to time constraints it was not integrated with the CGI system.

4.3 Strengths

The CGI system was extremely successful from the point of view of allowing anybody anywhere in the world to control a real robot, as shown by some of the creations in figure 4.6 on page 69. No software download, installation or configuration was required of the user, other than already being connected to the Internet and having a web browser installed. The interface used very few resources, and could therefore be used with low speed computers. The interface also limited forms of interaction meaning that it could be used with minimal prior knowledge of the system or even robotics. The communication mechanism (HTTP), being a standard web protocol, was accessible from almost all machines as HTTP is a trusted protocol that is not firewalled by system administrators.

4.3.1 Flexibility

A stated goal of developments of the system was to move towards a flexible/modular system that could be used in different ways, and for different applications. This was demonstrated in a number of ways.
4.4 Limitations

The template mechanism enabled users to design their own interfaces which enabled surprisingly different interfaces to be developed. It was used to provide a customised interface to a telerobot set up by Ken Taylor in the Carnegie Science Center, and was also used in Carter’s Lego lab cam project (Carter, 1997).

An unexpected aspect of the CGI interface was that the interface did not have to be rendered as a web page to be useful. Any application capable of sending and receiving HTTP Post requests could control the robot. HTML pages were still sent as replies, and needed to be parsed by the client, but as the format of these was determined via the template mechanism, these applications could effectively define their own communication format. This technique was used by Harald Friz for his Usher Java interface (Friz, 1998). It was also used by Chris Gunn from CSIRO, who wrote a CGI based system that interfaced with a Phantom input device to send positional commands to the robot. This was done with no communication or help from the telerobot group at UWA, and shows how open a CGI interface is (which may or may not be a good thing).

4.4 Limitations

Despite its success and popularity there were a number of outstanding problems with the system. Some of the problems areas included:

- Speed
- Session tracking
- State divergence
- No communication with other users
- No feedback during command execution
- Limited user interface capabilities

The speed limitation is due to a combination of factors. Firstly, there is significant dead time between an operator sending a request and receiving a response. During this time the browser window is either blank or frozen and the user has no feedback as to how the move is progressing, and cannot start to plan the next move. Another factor is that the complete page is updated, not just the bits that have changed which means more data must be produced and sent over the network and the browser must render the complete page every time. A final factor is that the browser must establish a connection with the server for each request, which adds further overhead to the overall processing time. If these factors are considered together it can be seen that the speed problem can be attributed to a combination of the request response nature of HTTP, and the page based interface provided by a browser.
The session tracking problem refers to identifying consecutive requests from the same user. As mentioned in the previous chapter, the system used was to insert a hidden field in every page sent to the user. This relied on the user using the same page for their next request. If their next request was via a different page without the correct hidden field they would be considered a new user. This often led to a situation where a user was told that they couldn’t use the robot because it was being used by someone else on their computer. Many strategies for session tracking have evolved over time, but all are difficult or troublesome to use. The fundamental reason is that as the HTTP protocol does not provide session tracking, solutions have to rely on tricks layered on top of HTTP.

State divergence occurs because the browser and CGI script are disconnected once a request has been processed. Any changes in state on the CGI side cannot be propagated back to the browser as HTTP does not provide server initiated communication. Additionally, the browser state can change without the CGI script being informed (the user pressing the back button for example). A further problem is that a submit button can be pressed more than once, leading the same request being sent twice. These unsynchronised changes in state mean that the state management within the CGI script is complex and error prone.

The telerobot system was very popular and there were often many users all vying to control the robot. One of the promises of web telerobotics is that all these users might be able to communicate with each other and collaborate. Ideally this service could be provided via HTTP and with the CGI script acting as a broker to relay messages between users. However, again the lack of server initiated communication means that the only time messages can be sent to a browser is in response to a request it makes.

As already mentioned, once a user sends a request their interface is frozen, and they receive no feedback until the command is completed. A more satisfactory solution would maintain the interface between moves, and only update the parts that have changed. For requests that take some time to process there should also be some form of intermediate feedback to indicate how the request is proceeding. Some of this can be approximated with the clever use of frames and multiple CGI scripts, but to do this requires a major contortion of the original design intentions of HTTP and HTML.

A final issue related to the user interface is that HTML provides very limited forms of user interaction. For example, in order to perform the stereo point algorithm, two HTTP requests had to be sent to the CGI script, as clicking an image always sends an HTTP request. This makes using this feature very slow, which is a pity as it potentially provides a more intuitive way of moving the robot than typing in coordinates. The actual stereo point algorithm is quite simple and could be performed locally with little client overhead. This was attempted with Javascript, but finding
a solution that worked reliably proved illusive due to implementation differences across different platforms.

4.5 Conclusions

The results presented in Taylor’s thesis show that the ABB 1400 system was a significant improvement over the Taylor96 system. This was shown from the increase in number of moves in a session and from the structures that people built with the system.

The partitioning and redesign of the system also showed a big improvement in the reliability of the system. The watchdog timer was no longer required and the distance moved by the gripper per week continued to increase. An interesting part of the redesign was the introduction of HTML templates that allowed users to design their own interfaces. This is something that is easy to do with HTML interfaces but although many websites allow users to customise preferences of the website, the author is unaware of any that have taken it this degree of allowing users to design their own interface.

Although the system was adjudged to be a big improvement over the original system, the various incremental changes made once it was online had little effect on how the system was used - the time between moves, and number of moves in session stayed consistent. Additionally, there were many problems associated with using HTTP and CGI - state management, and user identification being the two most important.

Some of these problems can be overcome with clever use of HTML extensions (such as Javascript, and frames) and HTTP extensions (such as persistent connections, and server push). However such solutions are still restricted by the original concepts of HTTP and HTML. To solve all these problems would require a combination of a number of different extensions, which makes the system more complex and harder to maintain. These extensions are not programming languages in the way that C++ and Java might be, and it is therefore hard to keep a structured and well partitioned design. Many of these extensions are also exposed to implementation differences across different platforms. A preferable solution would be to use a technology that still has the benefits of the web (an open cross platform standard, and simple distribution of the client interface), but one that is able to operate outside the confines of HTTP and HTML. Java is one such technology.
Network Protocols and Architectures

The CGI system worked well, and handled thousands of moves over a number of years. It provided an interface for controlling a robot over the Internet that was cross platform, required no specialist software or hardware, and was available to anyone with a web browser. However the use of the standard technologies of HTML (interface), HTTP (protocol) and CGI (server architecture) had its limitations as mentioned in the previous chapter. These can be summarised as follows:

1. Limited user interface interactions
2. Speed
3. Unreliable session management
4. No feedback during moves
5. State inconsistencies between client and telerobot site
6. No event notification

These limitations can be attributed to three components of the system. The interface (items 1, 4 and 6), the overall architecture (items 2, 4, and 5), and the underlying protocol (items 2, 3, 4, 5, and 6). Some of these limitations can be reduced to acceptable levels by the use of extensions such as Javascript(interface), HTTP push(protocol), and Java Servlets(server architecture). However as these solutions are extensions to the standard web technologies, they are not always implemented consistently across all platforms (an issue for client side techniques). In addition they are still based on the original web paradigm of a page based request response system.

An alternative solution is the use of Java Applets. An Applet is an executable that has access to a large number of underlying system functions, including both the user and network interface\(^1\). This means that both custom interfaces and protocols can be used allowing freedom to define exactly

\(^1\)Within certain restrictions, known as the Java sandbox
how the system will operate. Java Applets are still based around web technology and therefore maintain the advantages of a web interface, in that they are cross platform, require no specialist hardware or software, and are usable by anyone with a web browser\(^2\). This chapter considers protocol and architecture issues of a system based on using Java Applets for clients.

### 5.1 Protocol Design

CGI provides an application level interface that completely insulates the programmer from the underlying network protocol. As CGI abstracted the network, nothing was known about what type of network connections users had, and what the limitations and implications might be for a new protocol. Research was therefore undertaken to first understand the core network protocols used over the Internet and to then test the performance of these protocols from within Java Applets.

The following sections summarise the underlying protocols that are used on the Internet, and introduce terminology. Various models that have been developed to predict Internet performance are presented. A set of tests to measure some of these characteristics were developed and the results of these are presented and discussed. Finally, recommendations are made for the protocols and techniques that are most appropriate for Internet control.

### 5.2 Internet Protocols

Modern communication protocols tend to use the concept of layers. The lowest layer is the physical signal transport, middle layers provide various services such as routing, and the top layer provides a programmer interface. Probably the most well known standard for defining this set of layers is the *OSI Reference Model* (Day and Zimmermann, 1983), which has seven layers: physical, datalink, network, transport, session, presentation and application. Applications that communicate over the Internet use the *Internet protocols* (otherwise known as the TCP/IP protocols) to communicate. Although these protocols were not designed to conform to the OSI layering model, they can be retrospectively compared. The Internet protocols conform loosely to layers one to four of the OSI model. Of these, layer three the network layer, is provided by the *Internet Protocol* (Stevens, 1994a), and layer four, the transport layer, is provided by the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP) (Information Sciences Institute, 1981; Stevens, 1994b). As the Internet Protocols only provide services up to layer four, the sess-

\(^2\)The browser must support Java
tion, presentation and application layers are handled by the application as required. It should be noted that as the Internet Protocols only provide up to level four, and even this is incomplete (due to a lack of Quality of Service), they are relatively simple compared to true OSI protocols such as X.25 and asynchronous PSTN. This has made Internet Protocol based hardware and networks cheap to build, but this has come at the cost of being able to guarantee a quality of service. This is an important fact that web based systems must contend with. However the problem will begin to be alleviated with the introduction of the next generation of Internet Protocols (IPv6).

IP provides an unreliable, connectionless datagram delivery service. Unreliable means there are no guarantees that an IP datagram will make it to its destination. Connectionless means that it does not maintain any state information about successive datagrams. Each IP datagram consists of a set of headers and the data (also known as a protocol hole). The headers include the IP version, the length, packet identification number, and various other flags and parameters.

Layered on top of IP, are TCP and UDP that are used to provide an application interface. TCP is a connection oriented, reliable, byte stream service, while UDP is a connectionless, unreliable, datagram delivery service. In connection oriented services a connection must be established before communication can occur. One end point must request a connection from the other which must agree to accept. Once the two end points agree to communicate, the network system forms a data path called a connection. The connection stays in place until the either computer decides to terminate it. In connectionless services, to communicate, data is just packaged and addressed to the receiving computer, and passed to the network for transport. There is no need for a connection to be established.

Depending on the layer being considered, different terminology is used to describe the transported data units (Braden, 1989). At the physical layer (Ethernet) the data unit is an Ethernet Frame.
Between the Ethernet driver and IP layer it is called an IP packet. Between the IP and UDP layers it is called a UDP datagram. If it is between the IP module and the TCP module it is a TCP segment. If it is in the application itself then it is referred to as an application message.

Layered on top of TCP and UDP are Internet Application protocols. These include HTTP (used in the previous two chapters), and many other standard and non-standard services. Depending on the type of network application, a protocol may use the connection-oriented service provided by TCP, or the connectionless but unreliable service offered by UDP. Most protocols designed for use over the Internet (HTTP, FTP, Telnet) have tended to use TCP, as it handles flow control, and packet loss. Packet loss can be a frequent occurrence once packets must travel beyond the Local Error Network. However, error protection comes at a price, and applications that have needed performance with a LAN environment often use UDP, NFS for example.

TCP obtains its error protection by using an Automatic Repeat reQuest (ARQ) protocol. It automatically retransmits corrupted or lost packets to make such errors invisible to applications. When the transmission quality is poor this may cause significant delay and jitter. For interactive applications that require real-time responses this makes TCP less suitable (Miyata et al., 1998). Alternative schemes such as Forward Error Correction (FEC) send enough redundant information in the data stream so that it can be reconstructed even if some parts are lost. This is the preferred method for real-time audio applications. To implement FEC over the Internet, UDP must be used as this does not use ARQ.

An important property of the network layer is the Maximum Transfer Unit (MTU). This is the maximum size of packet that the network can send in one frame, if the route between two endpoints consists of more than one network link, then the MTU for the path is the smallest of the MTUs of each link. If the IP layer has a datagram to send that is larger than the MTU of the link, the datagram must be fragmented into smaller pieces. For Ethernet the MTU is 1500 Bytes (Comer, 1991a). The default MTU used by TCP is 576 bytes (TCP), which allows for 512 bytes of data plus the TCP and IP headers. TCP can perform MTU discovery for each connection, so that it always sends the maximum allowed segment.

5.2.1 Network Models

The two fundamental properties of networks are delay (or latency) and throughput (Comer, 1997). Delay has a number of constituents: propagation delay - proportional to distance; switching delay - network devices wait until all bits of a packet have arrived; access delays to shared media; and queuing delays when previous packets need to be processed first. Throughput is the rate at
which data can be sent through the network. The term bandwidth is often used as a synonym for throughput. More accurately bandwidth should be used to refer to the throughput of the underlying hardware. Most protocols include some form of header associated with each data packet, meaning that throughput is always less than hardware bandwidth. Throughput and delay are not completely independent, as under heavy network load (congestion) the delay is likely to increase. Delay has been shown to follow the following formula (Comer, 1997):

\[
D = \frac{D_0}{1 - U}
\]

where \(D_0\) denotes the delay when the network is idle and \(U\) is a value between 0 and 1 that denotes the current utilisation of the network. Once a network’s delay and throughput are known then the delay-throughput product can be calculated \(D \times T\) where \(T\) is the throughput of the connection. This gives a measure of the volume of data in flight over the network.

To model time taken for a message to travel between two end points, the simplest model (Paxson et al., 1998) for end to end time \(ett\) would be:

\[
ett = D + \frac{S}{T}
\]

where \(S\) is the amount of data to be transmitted. However this is a gross simplification of the underlying network path as it models it as a single stable link, instead of a multitude of routers, switches and links. It also takes into account the effect of \(S\) on the delay \(D\), the effect of temporal factors such as other Internet traffic, or possible changes in routing between subsequent packets. Other more complex models such as those proposed by Fiorini and Oboe (1997), model the connection as a network of queues each representing node in the network. Each queue is modelled as a FIFO queue with random arrivals and exponential service time (known as a \([M, M, 1]\) queue) (Hammond and O’Reilly, 1986). As the number of queues increases with the number of nodes traversed, the total delay is equal to the sum of an increasing number of exponentially distributed variables, which tends to a normal distribution. The delay distribution for typical Internet links has been shown by experimentation to be a normal distribution by a number of researchers: Bolot (1993); Sanghi et al. (1993); Oboe and Fiorini (1997).

These models help to give an idea of how a network connection might behave, but they are far too simple to model the realities of the Internet. As a result, all statistics should be treated with caution (Paxson and Floyd, 1997) as there is still no accepted way of predicting performance of a network.
link (Frogner and Cannara, 1999). There are also a huge range of capabilities of links from slow modems to fiber optic links with bandwidths a million times faster. Similarly some links have significantly more latency than others – 100’s of milliseconds for a geo-stationary satellite link.

In the past Telerobotic control has been performed using communication media that provides a guaranteed transfer rate, and constant delay. Networks that provide a small, predictable time delay are known as isochronous (Orfali et al., 1996a). These telerobot systems have tended to use closed loop fashion and have therefore required the delay to be predictable to avoid instability. Although some attempts have been made to perform closed loop control over the Internet (Brady and Tzyn-Jong, 1998; Fiorini and Oboe, 1997), the issue is best addressed by designing a system that is tolerant of variable time delays. Supervisory control is one such solution, as was the mode of control used by the CGI system.

When operating in a mixed environment, the more information about the current network conditions available to a protocol, the more efficiently it can use the network to transfer data (Ashir et al., 1999). Acquiring such information is particularly important for operation in wide-area networks, where a strong tension exists between needing a large amount of data in flight in order to fill the throughput delay product pipe versus having to wait lengthy periods of time to attain feedback regarding network conditions (Allman and Paxson, 1999).

5.2.2 TCP Operation

UDP provides a very thin wrapper on top of the datagram service offered by IP. By contrast TCP must implement the transport layer in order to provide an ordered reliable stream based connection. The core part of a TCP implementation is the windowing principle. This enables a sender to send multiple packets without waiting for a reply indicating that the first packet has been sent. The number of unacknowledged packets is determined by the size of the window. This is illustrated in figure 5.2 on the next page. The sender slides the window whenever a packet is acknowledged. In TCP the size of a sender’s window is configured by the receiver which indicates how much space it has left for receiving data. Corrupted and missing data are handled by the use of timers (one for each packet sent) and examination of the acknowledgement numbers from the receiver (see Stevens, 1994b, for more details).

To avoid unnecessary sending of small packets (tinygrams), TCP uses an optional congestion control algorithm known as the Nagle algorithm (Nagle, 1984). This algorithm prohibits small segments from being sent if the connection has outstanding data that has not been acknowledged.
Instead small amounts of data are collected and sent in a single segment once the acknowledgement arrives. The decision process used by TCP can be summarised as follows:

- If a packet is equal to or larger than the segment size (or MTU), and the TCP window is not full, send an MTU size buffer immediately.
- If the interface is idle, or the Nagle algorithm is turned off, and the TCP window is not full, send the buffer immediately.
- If there is less than 1/2 of the TCP window in outstanding data, send the buffer immediately.
- If sending less than a segment size buffer, and if more than 1/2 the window is outstanding, and the Nagle algorithm is being used, wait up to 200 msec for more data before sending the buffer.
5.2.3 What can be controlled by an Applet?

Most of these studies quoted so far have used known Internet hosts to perform their network test. This allows test software to be installed at both hosts. This means that some knowledge of the network is known a priori, and that some control may also be possible. The majority of work is concerned with the behaviour of a fixed packet size over known network link(s). These studies are primarily interested in optimisation of transport and network layer protocols.

A Java Applet communicates using an application layer protocol that is layered on top of UDP or TCP. Due to security and implementation issues Java does not expose the full TCP and UDP APIs, meaning that an Applet has limited control over how it communicates. However, it does have control of the following:

- Connection startup and termination
- Size of messages sent to transport layer
- Frequency of sending messages
- Which transport layer protocol to use (TCP/UDP)
- Whether the Nagle algorithm is on or off
- Send and Receive buffer size

To test how the optimum way of using these transport layers, a set of test Applets were designed, and distributed to users via the telerobot web page. These test Applets are the subject of the following sections.

5.3 Testing network behaviour

As the Telerobot client can be any Internet host, the exact network characteristics of the connection are not possible to predict in advance. However, tests of a representative selection of hosts can establish likely network characteristics. Characteristics that are of value, are whether connections can be established in the first place (firewalls and proxies may prevent this), end to end delay, and how this is affected by size and rate of sending of packets. By establishing these characteristics the protocol can be designed to maximise availability, reduce delay to a minimum, and maximise throughput. Measuring, end-to-end delay instead of round trip time is preferable as often the outward and return routes have different characteristics (Paxson, 1999). Unfortunately end-to-end delay relies on both peers having accurate clocks, and that they are synchronised. As the client is an unknown Internet host, this cannot be easily satisfied. Instead the round trip time is measured to give an estimation of end-to-end delay.
To perform the test on as many hosts as possible, and to use the same environment that clients were likely to run in, a number of network test Java Applets were developed. The Applets were distributed with a web page, that was downloaded as part of normal use of the robot. Users following a link to use the robot were asked if they wanted to assist in our research and perform the network tests. If they agreed, they were directed to the Applet test page which performed the test. On completion of the tests, they were then returned to the robot control page (still the CGI system at this stage).

The first Applets tested round trip delay for a sequence of packets of the same size. The bandwidth of a link is likely to change during the course of the connection (Paxson, 1999), as are the inherent latencies, so these tests were designed to gain an insight into average network characteristics of a link.

5.3.1 Round Trip Times - TCP

The first tests were performed with TCP. A server running on a machine at UWA listened on a port for connections from one of the test Applets. On establishing a connection, the server decided on the size and number of packets to send, and informed the Applet. The server then sent packets synchronously i.e. it waited for the first packet to be echoed before sending the next one. The Applet echoed all packets that it received until the expected number had been processed. Once the test was completed the server logged the results to a file. In some cases the Applet was not able to make the initial connection to the server due to firewall restrictions or other problems. To record these failures, the Applet logged successful and failed attempts via a CGI Post operation over standard HTTP. Given that the client was able to download the Applet using HTTP in the first place, it can be taken as given that the Applet was able to send information back over HTTP. The statistics logged by the Applet/server combination were: connection success/failure, connection establishment time, and time taken to send \( n \) packets of size \( x \). TCP performs round trip delay and bandwidth estimation as part of the protocol (Comer, 1991b), but this information is not readily available through the Java networking API. In addition this is the round trip delay of individual packets; a large application packet may be split into a number of TCP segments. As the tests were of network performance from the application layer it was more appropriate and easier to calculate round trip delay at the Java level. All tests were performed with the Nagle algorithm disabled. Each message was filled with random simulate data which makes any compression that occurs \textit{en route} more realistic.

Table 5.1 shows the results of trying to establish an initial TCP connection over an unprivileged port. 87% of clients were able to establish a connection. The connection failures were for a variety
of reasons. The most common error was an UnknownHostException, the exact cause of which was not established. It is unusual as the client must have been able to resolve the server’s hostname to download the Applet in the first place. In addition this the Applet is still able to make a HTTP Post connection to send the result back! A possible explanation is that HTTP connections are made through a proxy that resolves the hostname. Making a URL connection from a Java Applet would use this proxy, but DNS lookups within the Applet would not. Due to the sandbox security restrictions the hostname must be used by the Applet, and not the IP address so it is not possible to bypass the problem very easily. The other errors such as ConnectException and NoRouteToHostException are also all likely to be manifestations of different firewall/proxy combinations that block direct connections from a Java Applet to a remote server. To support these 13% of clients a system implementing a protocol that operates over HTTP would have to be implemented.

<table>
<thead>
<tr>
<th>Successes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failures</td>
</tr>
<tr>
<td>java.net.UnknownHostException: voyager.mech.uwa.edu.au</td>
</tr>
<tr>
<td>NoRouteToHostException: Operation timed out</td>
</tr>
<tr>
<td>ConnectException: Connection refused</td>
</tr>
<tr>
<td>NoRouteToHostException: Host unreachable</td>
</tr>
<tr>
<td>SocketException:</td>
</tr>
<tr>
<td>All</td>
</tr>
<tr>
<td>All</td>
</tr>
</tbody>
</table>

Table 5.1: Connections made from remote hosts. 87% of clients were able to open socket connections back to the server. The reason for the remaining 13% failing to make connections is most probably due to firewall and proxy configurations that prohibit them.

Figure 5.3 shows the time taken to establish successful connections. The distribution is sharply peaked with a median of 0.8 seconds, a first quartile of 0.6 seconds and a third quartile of 1.4 seconds. This is obviously a delay that should be avoided by holding open connections across requests. This can be easily achieved using a custom protocol, but may be harder using a technique such as HTTP Post, as although HTTP 1.1 makes provision for reuse of connections, this has not been implemented on all clients and servers.

The effect of application message size on round trip time is shown in Figure 5.4. These results were generated over a number of weeks from 1,500 tests run on 1,000 unique hosts. Each test consisted of sending and receiving of 50 TCP messages of a chosen size. The results are plotted on a log scale as there are large differences between some of the results. This is due to the many different bandwidths and latencies present in different network paths. With so many different network bandwidths and characteristics, a single trend or conclusion is hard to reach. However, it can be stated that connection speeds vary immensely, and that delay will increase with message size (as would be expected!). It would also appear that up to \( \approx 1,000 \) bytes the delay is fairly
5.3 Testing network behaviour

Figure 5.3: Time taken to establish a connection between a client Applet and server at UWA. The distribution has a median of 0.6 seconds and a 3rd quartile of 1.4 seconds. Connections should therefore be held open wherever possible to minimise waiting time.

constant, although this is probably exaggerated by the use of a log scale. However, messages below 1,000 byte are important as they represent smaller control and event messages that ideally should have minimal delay. If all tests below are considered, the median round trip time is 0.7 seconds with an upper quartile of 1.1 seconds, and a 95th percentile of 2.6 seconds. As this is round trip time, the single trip time can be assumed to halve this time which means the 95% of users will receive these smaller messages within just over a second.

5.3.2 Round Trip Times - UDP

The UDP tests used similar techniques to the TCP tests with a server listening on a port at UWA and Applets making connections from clients. UDP communication is different to TCP and therefore some changes to the characteristics logged were different. Firstly, as there is no permanent connection, between peers using UDP connection establishment time is not relevant. Secondly, delivery is not guaranteed, so an extra statistic of package loss was recorded. No separate connection was created for each client connection in a UDP server, so it was easier to manage the test
from the Applet side. The Applet first sent an empty packet to initialise the test and the server then replied with the number and size of packets to use. The Applet then attempted to send these packets to the server, which echoed them. After all the packets had been sent, the Applet posted the statistics using HTTP Post as with the TCP Applet. As the timing was now performed on the client side, the results are exposed to the different accuracies of client’s clocks. However, as the times involved are of the order of seconds per packet, typical computer clock frequency differences are likely to be insignificant (Almes et al., 1999). As packets can be lost in UDP, a timeout must be chosen after which a packet is assumed lost. This was fixed at 5 seconds as it was important to keep the timeout value as low as possible to minimise the inconvenience for users that had agreed to perform the tests. However, in retrospect this should have increased with packet size as larger packets are likely to take more time in transport. For data sets where the packet round trip time approaches 5 seconds this is likely to have artificially increased the loss rate. If no packets were received it was assumed that UDP communication was not possible due to firewall restrictions.

Table 5.2 shows the results of the UDP tests. Connections that achieved at least one send and receive of a packet are classed as successes. The majority of connections (68%) resulted in exceptions of which the main contributors were security exceptions due to a problem with Microsoft’s
### 5.3 Testing network behaviour

<table>
<thead>
<tr>
<th>Successes</th>
<th>Failures</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SecurityExceptionEx: cannot access 0</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td>InterruptedIOException: Receive timed out</td>
<td>679</td>
</tr>
<tr>
<td></td>
<td>IOException: error #014</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>SecurityExceptionEx: cannot access port 0</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>UnknownHostException: voyager.mech.uwa.edu.au</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>SecurityExceptionEx: Unable to check address 130.95.52.60</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>InterruptedIOException: peek timed out</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>NullPointerException</td>
<td>1</td>
</tr>
</tbody>
</table>

|           | 846                        | 1012  |

Table 5.2: Results of UDP test Applet. The majority (68%) of connections resulted in exceptions. The main contributor to these exceptions was a bug in the Java VM of Internet Explorer.

Java virtual machine. The issue (Jamison, 1998) causes a security exception to be thrown when an Applet tries to open a datagram socket. Although the bug has been acknowledged by Microsoft, a fix date has not been announced. The fixes suggested are to sign the Applet and make it trusted, or to use the Sun Java plugin. Even if this bug is excluded from the results then still 47% of connections failed completely. This high failure rate is probably due to the firewalling of UDP packets by many network administrators. End users are unlikely to have control over whether they can send and receive UDP.

UDP performance (figure 5.5) does not seem much better than that of TCP. However, there is significantly less data to analyse due to the large number of failed connections. The performance seems to be slightly better for larger packets. However, the main issue is that the large number of failed connections mean that a meaningful statistical comparison is not possible.

Figure 5.6 shows the reliability of UDP for those connections that succeeded. Only 58% of connections had zero failures, although 95% of connections had a failure rate of less than 25%. Any system that use UDP needs to allow for this lack of reliability. If the protocol must have reliable transport then it is generally more efficient to use TCP than to try and implement reliability on top of UDP. UDP is most useful for broadcasting a stream of updates, where subsequent data does not rely on previous information. These tests show that it is reasonable to expect most users to receive three out of every four packets.

Figure 5.7 shows the effect of UDP packet size on failure rate. This graph shows almost no correlation between packet size and failure rate. The slight increase for very large packets is probably attributable to the fixed 5 second cut off limit within the test Applet. This shows that there is probably no need to restrict the size of UDP packets in order to improve delivery reliability.
Figure 5.5: Effect of packet size on round trip time for UDP packets. The trend is much the same as that for TCP packets, although there is significantly less data available due to the high number of failed connections. 20 packets were sent and received and round trip time averaged.

The UDP tests show that for clients that are able to use it, UDP is reasonably reliable over the Internet and having a lighter weight protocol than TCP should be faster (the limited data available did not show a significant difference). However, only 30% of clients are able to use UDP successfully from Java Applets due to problems in Virtual Machine implementations, and firewall restrictions. A protocol layered on top of UDP therefore needs to have a fall back mechanism to a TCP based protocol. As the number of clients that can use UDP from Java is so small, implementing a UDP mechanism may not be worthwhile at all.

Round trip time per host

The previous tests showed how different hosts sending different sized packets performed over the network. However, each test was only for a single size of packet for a given host. To get a better understanding on how different sized application messages affected a given host and network connection, a new test Applet was written. This Applet sent a series of messages of different sizes
5.3 Testing network behaviour

Figure 5.6: Distribution of failure rate for UDP packets. This shows the unreliability of UDP, only 58% of connections have completely reliable connections, although 95% have a failure rate of less than 25%.

Figure 5.7: The effect of packet size on failure rate. This graph shows almost no corelation between packet size and failure rate, the slight increase for very large packets is probably attributable to the fixed 5 second cut off limit within the test Applet.

so that an overall picture could be obtained. To avoid temporal effects of the link, the size of packets was chosen randomly from a list of predefined sizes until all sizes had been tried.

Some results of these tests are shown in figures 5.8 on the following page and 5.9 on page 93. The left hand plots show the average transfer rate, and the plots on the right show average delay, both plotted against message size. The plots show results from hosts in Australia, Europe and America.
Many other tests were made but these were chosen as a representative sample. The connections exhibit different characteristics as would be expected.

All connections show a delay that is fairly constant up to a point. This is due to the physical latency of the link and message processing overhead. Once messages are over a certain size, then
the round trip time starts to increase, as the message has to be split into a number of separate packets to travel over intermittent links. The size at which this transition happens varies from 800 Bytes to 5,000 Bytes.
The tests operated by waiting for each send and receive pair to be completed, before starting the next cycle. This therefore uses only half the network capacity as either the forward or return path will idle while the other is busy. However, the number of bytes sent and received per second still gives a measure of how efficiently the link is being used. In almost all cases, the throughput rises, to a point and then levels off. In some cases it even dips. However, as a general rule it would seem that larger messages use the network bandwidth more efficiently.

**Buffering Data**

A final set of tests was designed to investigate the effect of buffering data, over sending it as it arrived. The size of the data to send and receive was chosen as 1260 bytes, as this is the nearest highly composite number to 1000 bytes. This represents 1 seconds worth of data in an application producing data in small packets at a rate of 1Kbyte/s (the maximum throughput shown by some connections). The tests could then be used to see what effect buffering the data and sending it less frequently would have on the round trip time.

For each connection, a factor size (36 in total) was chosen as the unit message size. The message was then sent using the unit size until all data was sent. The Applet then echoed this back to the server. The process was repeated for each factor size. The sequence of sizes was chosen randomly to spread temporal changes in the link characteristics across the dataset. For example, the test might first start with 315 messages of 4 bytes each, then 84 messages of 15 bytes, etc. until all factor sizes have been tried.

Figure 5.10 on the facing page shows examples of results obtained from the test. From these graphs it can be seen that sending data in small messages, is significantly slower than buffering it and sending larger messages. This is true even on a LAN (the first graph). In all cases sending data in its atomic form (1 byte messages) increases the time taken by a factor of ten. This is not surprising as each message sent by TCP must have both TCP and IP headers added to it (64 bytes), so sending a message in \( n \) parts will add \( 64 \times n \) bytes to the size of the data. Each of these messages must be acknowledged as well, meaning that an extra \( n \) acknowledgements are required. These graphs show that buffering data at the application layer is essential and that messages below 100 bytes long should be avoided. Although the Nagle algorithm can assist with this to an extent, its maximum buffering time (200ms) may not be long enough in some cases.
5.3 Testing network behaviour

Figure 5.10: Buffering data before sending it has a significant effect on round trip time. Buffering 1 seconds worth of 1Kbyte/s data results in a factor of 10 reduction in round trip time over sending data as single byte messages.

5.3.3 Conclusions from Network Tests

Although Applets do not have as much control over network performance as native applications, there are still some important implications for communication over the Internet. The following conclusions can be drawn from the above tests:
• From within an Applet, TCP is more likely to succeed than UDP. However in some network configurations the only TCP port/protocol that will be allowed is port 80 using HTTP.

• An application should maintain permanent connections to avoid the repeated setup cost of creating new connections.

• Different connections have very different properties, factors of 10 or even 100 are involved. It may therefore be appropriate to provide data at different rates depending on the type of connection a user has.

• Where possible Data should be buffered before being sent, as this allows TCP to make the most efficient use of the network layer. How long to buffer for should be determined by the maximum acceptable latency, and the round trip delay for the link as used by the Nagle algorithm.

Firewalls - an aside

For a network application to succeed, it must be allowed to make connections from client to server. If this is blocked by a firewall, then the application will either fail or must fall-back to a more common network port that is not likely to be blocked. The current status quo is that HTTP is open on most networks, so a network application must resort to tunnelling its data over HTTP. Given time and effort, most protocols can be made to work over HTTP with an efficiency tradeoff. Therefore, the firewalling problem moves from a port based to application based. The firewall must understand the contents of the protocol and filter it accordingly. This is a potentially recursive problem between levels of tunnelling and levels of firewall protection. Each level of tunnelling adds size to messages, and therefore reduces the efficiency of the protocol, and ultimately the network that it operates over. Taken to the extreme this would lead to a situation where all traffic is tunnelled over HTTP, taking up far more bandwidth than before. These messages are then processed by a complex application level firewall, which will undoubtedly be slower, before letting the message through to the destination machine.

It should be noted that for most applications, making outgoing connections is all that is required by a client. The server is the only part of the network application that must handle incoming connections. System administrators can therefore allow outgoing connections on most ports, while treating incoming connections as carefully as they need. This will allow most applications to run over the Internet while still securing the local network from malicious incoming traffic. On the other hand if outgoing connections are blocked, then smart applications will tunnel over HTTP, leading to the above mentioned problems.
5.4 Web Telerobot Requirements

The network tests discussed in the previous section show the best ways to use the transport layer from within an application. This is important in improving the speed of the system, but this still leaves a number of unresolved problems with the CGI system. Some of these are problems are associated with HTTP (the application protocol) and some were problems with CGI (the system architecture). To address both application protocol and system architecture issues a framework is required. This framework should provide the same functionality as the CGI system, but also address the CGI limitations. These can be collected together into a set of functional requirements for a web telerobot framework:

**Connectivity**  An application must be able to make connection(s) in such a way that it is able to communicate with any other parts of the system. Users may need to communicate with other users; the robot may need to communicate with a user, etc.

**Authentication**  Authentication means that users of the system have some form of unique identification, this identification is determined by the exchange of credentials such as a username and password. Once authenticated, a user’s details can be retrieved. This may include information such as personal preferences, authority level, group membership, personal information etc.

**Session Management**  When a user or agent is authenticated with the system they start a new session. This session may well persist beyond the lifetime of the connection. Session management enables tracking and identification of different users. Session tracking is essential when resources are allocated to a particular client. All requests for the resource must be identified and only the correct client allowed access.

**Addressing**  Addressing allows one part of the application to communicate with another part elsewhere. A scheme must be provided to allow parts of the application to establish these addresses.

**Access Control**  Parts of the system might only be accessible to certain users, administrators for example. Access control allows control over who has access to different parts.

**Resource Control**  Physical Resources such as a robot cannot be easily shared, so control must be restricted in some way. Therefore there needs to be a way of establishing control/ownership and notifying other parts of the system when this changes.

**Asynchronous Communication**  Changes in state or events need to be propagated to other parts of the system, this requires asynchronous communication. Some asynchronous messages
may require acknowledgements/replies, in which case there must be a way of identifying the originating message in a reply.

**Synchronous Communication** Some commands require a reply in which case synchronous communication mode can be used. In synchronous mode a client makes a request and waits for a reply before continuing.

**Multicasting** New events such as a robot reaching a new position should be broadcast to anyone that is interested, not just a single client. There needs to be a mechanism by which consumers can register for events, and a sending mechanism that allows data to be broadcast.

**Reliable Delivery** Some messages (commands, and their replies for example) must be reliably delivered to their destination. The protocol must therefore provide reliable delivery that notifies the sender of any failures.

**Ordered Delivery** For messages that are dependent on each other the order in which they are processed is important. These are normally commands that have some temporal or spatial relationship to the state of a device, or its environment. For example, a relative command, move left, is issued on the assumption that the device is in a particular place. Whereas an absolute command such as move to 220 degrees and take a photo for a camera, has no temporal or spatial dependence on previous commands.

**Stream Separation** A consequence of ordered delivery is that a large messages can hold up other messages as they must be delivered first. Ordered message delivery is only needed for logically related streams, i.e. all images from camera 1, or all robot commands. The protocol should therefore allow these logical streams to be delivered independently.

**Message Prioritisation** Some messages within a logical stream may have higher priority than others and should be processed first. The most obvious example is the requirement to stop all pending commands due to an error. A stop command must be able to overtake currently queued commands based on its priority.

**Adaptive Bandwidth Use** If a client has a slow connection, then they may not be able to receive all data that is being produced within the system. The amount of data sent to client must be reduced by either discarding some of the data or switching a lower bandwidth source. There must be a method by which important messages are always delivered, while other less important ones are able to be discarded.

**Protocol Independence** Depending on the network conditions, the underlying transport protocol might use TCP, or UDP, or some other protocol layered on top (i.e. HTTP tunnelling). The application protocol should be able to use these different protocols transparently.
Delay Management Time delay management is important for any control application. Generally, the largest portion of the delay is caused by data travelling over a network. This is determined by the network path connecting parts of the system, and there is little scope for control of this. Even though the delay cannot necessarily be controlled, it can at least be measured, providing estimates of just how old information is. This allows data from different sources to be synchronised.

Platform and Language Independence A Telerobot system is likely to consist of a number of subsystems, that may operated on different operating systems, and be written in different languages. To allow these different applications to communicate, a protocol must address the data representation problem.

Web Enabled The protocol must work within a web environment. This means it must be able to operate from within an unsigned Java Applet. It should use TCP to communicate but should also have a fall-back mechanism to HTTP (over TCP), for fire-walled environments. It must also require no specialist software on the client, and be free of software license fees.

Flexibility A telerobot system is likely to change over time, due to changes in both hardware and software. The protocol needs to be flexible enough to include new systems and functionality without any change.

Synchronisation States changes in parts of the system need to be distributed to all clients, so that the distributed state remains synchronised. This requires asynchronous communication, multicasting, and possibly delay management.

5.5 Distributed Architectures

There are a huge number of distributed computing environments that all go some way to providing the requirements mentioned above. That is because many of them are requirements that are common to many distributed computing and collaborative applications (Schooler, 1996).

The one requirement that very few environments were able to satisfy was being Web Enabled. This is in fact a very restrictive requirement as it requires the solution to be small, and free. It must require no additional installation on the user’s part while providing all the other requirements. At the time this research was undertaken (late 1998), the author was unable to find any distributed computing solution that satisfied the above requirement.

This section considers some of the distributed systems technologies normally used with Java and analyses them against the list of requirements mentioned above.
5.5.1 CORBA

CORBA uses an Object Request Broker (ORB) as the middleware that establishes a client-server relationships between objects, and is an object orientated extension of Remote Procedure Calls (RPC) (Sun Microsystems Inc., 1998). Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object’s interface. The protocol for executing a call is defined through the application interfaces via a single implementation language-independent specification, the Interface Definition Language (IDL). CORBA operations are inherently synchronous as they are based on RPC semantics. CORBA does provide a event service that can be used for asynchronous communication although this is not implemented by all ORB vendors.

CORBA is primarily an RPC based system with asynchronous services, it is also one of the most sophisticated and complex distributed system environments available. This means that is requires a large supporting library to operate. Although CORBA support is supplied with Netscape, it is not included with Internet Explorer. This effectively rules out CORBA as a solution at this stage as the required libraries will not be installed on the majority of user’s machines. If CORBA was supported on all browsers it probably be the distributed application of choice for web Telerobotics.

5.5.2 Remote Method Invocation

Remote Method Invocation (RMI) is a Java specific Remote Method Invocation middleware, it provides a simple and direct model for distributed computation with Java objects. This simplicity is enabled by restricting all communication to Java objects only. There are solutions to running code in other languages but these involve writing a Java wrapper to the original application using the Java Native Interface (JNI). RMI uses a similar concept to CORBA’s ORB to provide remote object lookup and invocation. RMI has an added advantage that due to its pure Java implementation entire objects can move between client and server allowing full object orientated polymorphism.

RMI provides synchronous communication only. Asynchronous calls must therefore be executed synchronously as remote procedure calls, this makes message passing very inefficient and slow as a server must wait for a client to finish processing a message before it can continue. Like CORBA
5.5 Distributed Architectures

RMI is not supported by Internet Explorer so again will not work in a large number of browsers. RMI is therefore never likely to be a good solution for web Telerobotics.

5.5.3 Message Oriented Middleware

MOM unlike CORBA and RMI is not an industry standard, it is a collective term for a specific class of middleware that supports the exchange of general-purpose messages in a distributed application environment. Data is exchanged by message passing and/or message queueing supporting both synchronous and asynchronous interactions between distributed computing processes. The MOM system ensures message delivery by using reliable queues and by providing the directory, security, and administrative services required to support messaging. Queueing is particularly useful for processes where each step is dependent on the last.

MOMs model messages in an event delivery system, instead of as method calls. A MOM may present directory services that let clients look up another application which is acting as a server, or it may present all purpose channels that let a group of clients communicate as peers. The MOM itself is only a message router. Synchronous calls and other more sophisticated services can be built on top of the asynchronous base. MOM based systems have the advantage of being very lightweight and therefore are well suited to Applet distribution.

5.5.4 Java based frameworks

There are a number of Java based frameworks designed for developing distributed systems. These include the Java Shared Data Toolkit (JSDT), IBM’s Shared Data Objects (SDO), Habanero (Chabert et al., 1998), Infospheres (Chandy, 1996) and Tango (Beca et al., 1997), all of which come quite close to satisfying the requirements mentioned above. However they are all limited in one or more of the following ways:

- Not Web Enabled - They require additional software to be installed, or are not free. Despite being written in Java supporting Java Applets does not seem to be priority for these kind of frameworks. One further restriction is that some of these technologies are not freely distributable and cannot therefore be included with Applets.
- Not Platform and Language Independent - although these frameworks are all written in Java and are therefore able to run in many platforms, they all make use of Java’s serialisation to
transport objects over the network. This means it is very hard to use other languages as the protocol is language specific.

### 5.5.5 Connection Architectures

Distributed systems can be connected together in a variety of ways. To connect all the parts of a distributed system, the most obvious solution is a connection between each part/peer. However, this can quickly become complex as the number of peers increases, the number of connections required for \( n \) peers is \( \frac{n(n-1)}{2} \) as shown in figure 5.11. Each peer must also know the location of, and be capable of making a connection to, any servers or peers that it needs to communicate with. If peers or connections are unavailable the peer must be able to handle each appropriately.

![Diagram of Peer to Peer connections](image)

**Figure 5.11:** Peer to Peer connections, the number of connections can quickly become unacceptable, and adding a new connection at a later stage may involve a redesign of all peers

A more flexible approach is to use a central server or router that each peer connects to, to establish indirect contact with other peers as shown in figure 5.12. The router reduces the total number of connections in the system significantly to only one connection per peer \((n)\). Peers then only need to know the location of the router to communicate with other peers, plus access to other peers can be controlled centrally. The disadvantage of this approach is that the central server must be running and accessible to all peers, and that all information must flow through it. Thus the central router must be reliable and capable of handling large amounts of data.

![Diagram of Central Server connections](image)

If parts of a distributed system are grouped in close proximity, and communication to the central router is slow, performance can be improved by the use of local proxies that reduce communication with the central router by implementing part of the router’s functionality, however this has the disadvantage of increased complexity.
5.6 Conclusions

This chapter has covered a number of areas of developing a distributed web telerobotic system which included: tests on different transport protocols; identification of a set of functional requirements for a framework; and a discussion of some already available frameworks.

The tests on transport protocols established that TCP was preferable to UDP due to firewall and JVM implementation issues. TCP connections from different hosts exhibit performance differences that can differ by several orders of magnitude. It is therefore important to allow for these different capabilities when building a distributed telerobotic system. The tests also showed that not buffering data can have disastrous effects on overall delay, and that for slower connections it is probably advisable to buffer data for longer than the 200 ms provided by the Nagle algorithm.

A set of functional requirements was established that a web telerobotic protocol/framework should provide. This represents a list of services that are likely to be required by most telerobot systems which can be summarised as:

- Connectivity
- Authentication

Figure 5.12: Peer to router architecture with possible application to a Telerobotic system. Only one connection per peer is required and no knowledge of the location of other peers is required. Peers can exchange information with any other peer connected to the router.
• Session Management
• Addressing
• Access Control
• Resource Control
• Asynchronous Communication
• Synchronous Communication
• Multicasting
• Reliable Delivery
• Ordered Delivery
• Stream Separation
• Message Prioritisation
• Adaptive Bandwidth Use
• Protocol Independence
• Delay Management
• Platform and Language Independence
• Web Enabled
• Flexibility
• Synchronisation

Unfortunately no distributed frameworks were found that satisfied this set of requirements, and it was therefore decided to develop a reusable framework that would not only be applicable to the UWA telerobot but also be useful for other web telerobot systems. This would be based on the peer to router MOM architecture and is the subject of the following chapters.
Chapter 6

A First Step Away from CGI

This chapter was originally written in October 1999 and published in the June 2000 issue of the IEEE Robotics and Automation magazine (Dalton and Taylor, 2000). The paper is included here as it appeared in the magazine. It describes the first design of distributed architecture to support a distributed robotics over the Internet. The architecture is designed with the expectation that Java based clients running as Applets will be a significant component of the system. This means that the code size must be small, while providing flexibility for different applications, and working within the confines of browser imposed limitations on Applets.

Some of this work predates the network tests described in the previous chapter.

6.1 Introduction

Internet based telerobotics enables cheap, flexible, expandable and truly distributed systems to be implemented. Standard Internet hardware can be used as the basis for communication, and standard computer hardware can be used to run the client interface. The location and hardware capabilities of the operator(s) need only be determined at run time. Any number of robots, agents and operators connected to the Internet can communicate and interact together to achieve remote tasks. This was exploited by NASA’s Pathfinder mission. Initially scientists had to travel to the control centre in California to program the robot on Mars, despite the fact that the link from Earth to Mars had limited bandwidth. An Internet interface (Backes) was developed and it became possible for scientists to collaborate on and control the Pathfinder mission from anywhere in the world (Backes et al., 1998). This provides a powerful example of some of the possibilities of Internet control.
To enable control and collaboration between multiple users and agents, a communication protocol must be established. As this protocol must be used by all parts of the system, it is also useful to have a reusable software framework that implements this protocol. This paper describes a flexible and extendible protocol/framework that has been developed to control the UWA telerobot. The framework allows multiple peers to connect through a central router. Peers exchange messages through communication “channels” choosing to subscribe to channels they are interested in. The router has no knowledge of the content of these messages other than their intended destination and channel. The dynamics and behaviour of the system is thus determined only by the peers connected to it.

This paper first discusses standard Internet protocols and extensions known as middleware. Different middleware technologies are compared in the context of Internet telerobotics. A protocol and framework suitable for collaborative telerobotic control is introduced and discussed. Finally an example of how the framework might be used for a simple telerobot system is presented. The system has been tested locally but is not yet freely available on the Internet.

6.2 Internet Robotics

With the rapid growth in the Internet there has been a similar increase in the number of communications technologies available to execute requests in a networked environment. The most widely used of these being the HTTP (Gettys et al., 1997) protocol used by all of today’s Internet Browsers.

HTTP can be combined with CGI (The Common Gateway Interface) to enable remote process execution. This is one of the methods currently used to control the UWA telerobot (Taylor and Trevelyan, 1995),(Taylor and Dalton, 1997) and many other Web telerobot systems such as the Mercury project (Goldberg et al., 1995a). CGI produces dynamic content by launching a process to create a new page for each HTTP request.

CGI combined with HTML has a number of limitations. A new process must be started for each user request, causing latency and poor scalability. There is no support for server-server, (server initiated) server-client or client-client communication, which is problematic for robotics applications where the server needs to play an active role. Although HTML supports some simple widgets, it is not a user interface toolkit and the interaction styles which can be supported are very limited.
These limitations can be overcome by the use of Java. Java enables a client to have control over network connections and have a fully functional user interface while still operating within the confines of browser technology (browsers run Java programs as Applets). Being able to control network connections means that any protocol can be implemented, so that server-client, or client-client communication can be used. The ability to make permanent connections from a client to other servers and clients (peers) results in a “distributed system” allowing interaction and collaboration between any inter-connected peers. Distributed systems involving robots are often termed Distributed Robotic Systems (DRS).

Some examples of previous work Distributed Robotics Systems include Paolini and Vuskovic (Paolini and Vuskovic, 1997) who developed a distributed robotics lab, and Burchard and Feddema (Burchard and Feddema, 1996) who developed a distributed supervisory control scheme for control of devices with up to six degrees of freedom. Different Distributed Robotics Systems use different communication technologies such as CORBA, RMI and MOM(Orfali et al., 1996b; Shoffner, 1998) to achieve distribution. Below is a brief description of some of these technologies otherwise known as middleware (Orfali et al., 1996a).

### 6.3 Middleware Technologies

There are a large number of middleware technologies to choose from, each offering different features. The following list is by no means comprehensive; it is a subset containing those most relevant to Robotics over the Internet using today’s technology.

#### 6.3.1 CORBA

CORBA uses an Object Request Broker (ORB) as the middleware that establishes a client-server relationship between objects, and is an object oriented extension of Remote Procedure Calls (RPC)(Sun Microsystems Inc., 1998). Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object’s interface.
6.3.2 Remote Method Invocation

Remote Method Invocation (RMI) is a Java specific Remote Procedure Call middleware, it provides a simple and direct model for distributed computation with Java objects. This simplicity is enabled by restricting all communication to Java objects only. RMI uses a similar concept to CORBA's ORB to provide remote object lookup and invocation. RMI has an advantage that due to its pure Java implementation, entire objects (instead of just their data) can move between client and server allowing full object oriented polymorphism.

6.3.3 Message Oriented Middleware

Message Oriented Middleware (MOM) unlike CORBA and RMI is not an industry standard, it is a collective term for a specific class of middleware that supports the exchange of general-purpose messages in a distributed application environment. Data is exchanged by message passing and/or message queueing supporting both synchronous and asynchronous interactions between distributed computing processes. The MOM system ensures message delivery by using reliable queues and by providing the directory, security, and administrative services required to support messaging. Queueing is particularly useful for processes where each step is dependent on the last. MOMs model messages in an event delivery system, instead of as method calls.

6.3.4 Comparison of Middleware

MOM supports messages and is therefore primarily designed to support deferred asynchronous communication while RMI and CORBA, which are based on remote procedure call (RPC) semantics are designed to support synchronous communication. Message based communication also allows broadcasting of information to multiple peers. Both RMI and CORBA are standards and for that reason offer greater interoperability potential, although RMI has the restriction that all applications must be written in Java. However CORBA and RMI support in todays browsers is not universal due to third party interests and versioning differences. MOM on the other hand can provide a lightweight client API that can be included with the Applet. Shoffner (Shoffner, 1998) lists some of the advantages and disadvantages of MOM verses RMI and CORBA showing that MOM offers most potential in simpler environments that do not require synchronous communication and where installation of prerequisite software may be a problem. In more complex environments CORBA or RMI should be considered.
Telerobotics is an application where non-blocking asynchronous behaviour is actually desirable. Remote execution of commands can take a number of seconds and there are likely to be intermediate results. This behaviour is only readily available in a MOM based system. MOM’s semantics fit with those of *supervisory control*, where control signals and results are seen as messages between controller and manipulator. It is important that messages are executed in order, and that each is dependent on the last - this again fits well with the MOM framework. MOM is also especially well suited to situations in which clients collaborate as peers. For these reasons MOM was chosen as the basis for the distributed application framework.

### 6.4 The Framework

The framework uses a central MOM based router to transport messages between peers shown in figure 6.1. The concept of Client/Server is replaced with the concept of peers cooperating together to achieve tasks. Hence the term peer can apply equally to code controlling the physical robot, or to code on a personal computer displaying images of the robot workspace.

![Figure 6.1: Peer to router architecture. Only one connection per peer is required and no knowledge of the location of other peers is required. Peers can exchange information with any other peer connected to the router. Messages are sent between peers in channels that peers subscribe and unsubscribe to.](image)

#### 6.4.1 The Router

Peers connect to the router using a TCP/IP socket at which point they are authenticated. Once logged in, the peer can then publish and subscribe to different communications channels within the router. These channels are distinguished by name. Messages are routed between peers based
on who the message is addressed to, and what channel it is for. Addresses are passed as an array of strings with a zero length array indicating the message should be broadcast. Peers must implement the actual message handling for each channel.

The router also provides a naming service that enables peers to look up other peers and find out who is providing a service for a given channel. Access control to a given peer/channel can also be controlled by the router if the peer so desires.

To use the Router’s services a peer must adhere to a simple message protocol shown in figure 6.2. It includes a message start character, message size, channel, destination, message identity, reference identity, and priority. The structure and implementation is based on that outlined in Hughes et Al (Hughes et al., 1997). Format for individual fields in the header are either 4 byte integers or Java’s UCS transformation format (UTF) strings.

![Message structure for sent and received messages. All messages must start with a designated start character. Strings are sent in the Java UCS transformation format (UTF). The router examines the To address(s) and channel only, the To address is replaced with the sender’s address before being forwarded to the relevant peer(s).](image)

In order to process a message the router first decodes the destination and channel. The destination is then replaced with the identity of the sender and forwarded to the appropriate peer(s) message queue. The router maintains a separate thread for each peer that processes its own queue periodically.

Messages sent addressed directly to the router are interpreted as local commands. These include commands such as subscribe, unsubscribe, lookup service, and gain access to restricted peer. Access control for a given peer/channel is specified by the peer at login, the default being free access. Peers can choose to let the router manage access control for them, choosing from a number of access control algorithms. These include a priority/time out algorithm that restricts access to a single client, or a “tied” algorithm that ties access control to that of another peer/channel.
6.4.2 Peers

Peers must send and receive messages wrapped with the basic message structure of figure 6.2. Each peer is implemented as a set of software clients that connect through a central “router connector”. Each client must specify the name of a channel that it is designed to handle. The channel may be specified at design time or obtained through a service lookup. To send a message a client passes the message to the router connector. The router connector attaches the appropriate channel label, and other message headers including the message size before sending the message to the router. Similarly the router connector checks incoming messages for correct message size, demultiplexes the channel, and forwards them to the appropriate client. Clients are free to implement their own internal message protocol without any need to understand the router’s wrapper message structure.

The fields, identify and reference identity, can be used by clients to identify replies to messages so that synchronous communication is possible. Typically a client will keep a list of actions to perform for each message identity it is expecting a reply to. When a message with the correct reference identity is received the action is performed and removed from the list. Priority can be used by clients to implement priority based queues where high priority messages can overtake less important ones. This is particularly important for control of a robot where a number of outstanding control messages may need to be side-lined in favour of an important message such as “STOP”.

The architecture of a peer is shown in figure 6.3. A number of clients (image and robot clients in this case) all connect through the “router connector” to send and receive messages. Although these are termed connections, in reality the peer is usually a single executable, in which clients and the router connector communicate using function calls.

![Figure 6.3: The architecture of a peer. Each client within the peer subscribes to a different channel. Messages to and from each client are multiplexed and demultiplexed by the “router connector” which handles all communication with the outside world.](image)
A generic router connector and client have been implemented in both C++ and Java. The C++ version uses the Adaptive Communication Environment (ACE) (Schmidt and Suda, 1994) from the University of Washington, which is a multi platform framework for concurrent communication. Thus code written using ACE can be compiled without change on most operating systems. The Java code extends the client and connector of Hughes et al (Hughes et al., 1997) to include extra message flags and additional router commands. Due to the cross platform nature of Java, this may also be used on any platform supporting Java without change.

Creating a new peer only requires implementation of clients for new channels that are to be handled. Clients can be derived from an abstract client in either C++ or Java. The “router connector” itself is generic and can be reused without change.

6.5 The Framework in Practice

The framework has been used to implement control of the UWA telerobot using Java Applets distributed with an HTML page. The system currently comprises of two control peers, the router itself, and any number of Java peers operated by users (see figure 6.4). One control peer, the robot peer, communicates with the robot controller. The other control peer, the image peer, takes images of the workspace using a number of cameras.

The robot peer provides a robot control and robot state service, each on a separate channel. The control channel is access controlled by the router using the idle time algorithm. The state channel is open access and broadcasts the state of the robot at regular intervals. The state of the robot consists of the robot’s pose, the gripper position, and execution mode.

The image peer provides an image control and image data channel. Access to the image control channel is tied to the robot control channel, so only the current robot operator can change size, quality and type of images produced. The image data channel is free access, and broadcasts information on new images whenever they are taken. A third client in the image server subscribes to the robot state channel, and listens for new robot states. On receiving a new robot state images are taken of the workspace and the image data channel is instructed to broadcast the new image details.

Users can download an Applet that subscribes to both the robot and image channels. A user can try to gain control of the robot control channel, and if successful, can send control commands to the robot via the robot control channel. The user can also change image size and quality for images
Figure 6.4: The peer and router architectures shown for a single JAVA client, robot, and image capture device. The robot server subscribes to the robot channels and executes move requests as they are received. Access control is managed by the router. New robot states are broadcast by the robot server as the robot is moved. The image server subscribes to the robot state channel, and takes images for each new robot state, these images are then broadcast on the image channel. The Applet subscribes to the robot and image channels to receive updates and to submit control requests.

of the workspace via the image control channel. Status and position of the robot are received asynchronously over the robot state channel during task execution, as is new image data over the image data channel. If a user fails to control the robot, then they can still receive robot state and image updates via the robot state and image data channels. The Applet also subscribes to a chat channel so that users may communicate with each other while controlling/watching the robot. An example of the Applet running is shown in figure 6.5, this is a screen-shot from a second test system setup at ABB Corporate Research, Norway.

The Applet interface shown in figure 6.5 shows a four move session controlling the robot. Robot commands are entered and monitored using the text fields on the left side of the interface, while the right hand side shows current images and messages sent by other users. To move the robot the user must first gain control, then create and send robot commands. Robot commands can be
Figure 6.5: Demonstration Applet controlling a IRB 2400 robot at ABB Corporate research, Norway. The interface consists of dynamically updated calibrated images, current and past robot state, and a chat interface. All parts of the interface use the central router to communicate with other parts of the telerobot system.

entered directly, or produced dynamically using an Augmented Reality cursor (Dalton et al., 1999). As commands are executed their status is updated in the robot command section of the interface, along with any errors. Full details of errors are provided in the console area at the bottom left of the Applet. Visual updates are provided as new images are obtained from the image peer. New images are overlaid with the current robot position. As sending robot position consumes significantly less bandwidth than images, the overlaid robot position is often ahead of the actual image.

This example illustrates the dynamic configurability of the system, applied to the control of the UWA robot. The behaviour of the system is determined by which peers are present at run time. It also shows an (admittedly simple) example of collaboration between the robot and image peers to provide image and robot updates to the user. If a number of users are running the Applet they can
use the chat channel and robot channels to share control and exchange ideas and hints. However, the potential for collaboration has not been investigated fully, as the system has only been tested locally and is not yet available on the Internet.

6.6 Future Work and Conclusion

The example of a use of the framework shown above only begins to test its capabilities, configurability and ability to enable collaboration. There is some collaboration between the robot and image peers, but little between the Java clients other than the chat channel. Future work will concentrate on establishing collaboration between human operators using the Java peers and other peers. This will include a workspace blackboard allowing users to post location and types of objects to a workspace manager. Users will then be able to work together to interactively build models of the workspace to assist in teleoperation of the robot. This will be part of parallel work researching interactive workspace modelling in telerobotics. It is also hoped that other robotics groups may provide collaborative peers that may perform tasks such as image processing, robot path planning, and robot command interpretation.

The framework has been implemented and tested using a number of Java peers, a robot peer and image peer, all connected through the central router. It has proved capable of allowing flexible communication between arbitrary clients. The peer software is suitably generic to be usable on almost any platform in either C++ or Java. Implementation of new peers, only requires implementation of new channels, as the router connection is handled by generic code. New clients to handle channels can be derived from a base Java or C++ class.
The design outlined in the previous chapter was implemented and tested locally. It was successfully used to connect C++ and Java peers together to exchange messages and control the robot. This enabled different parts of the system to be designed and built independently with very little coupling. At the same time different parts of the system could easily communicate with each other via the MOM. Implementing applications was much simpler as user authentication, multiple clients, their connections and resource allocation were all handled by the MOM.

However, the design did not satisfy all the functional requirements listed in chapter 5. It also became apparent that there were a number of limitations in its design, which manifested themselves as a lack of flexibility. The following requirements were identified as being either unimplemented, or implemented badly.

**Protocol Independence** Only one transport protocol was implemented (the fixed header protocol) and there was no provision for using alternative protocols.

**Adaptive Bandwidth Use** This was not implemented.

**Time Delay Management** This was not implemented.

**Access and Resource Control** Although implemented, both access and resource control were tied together by the use of the access controlled channel. These two concepts needed to be implemented separately.

**Flexibility** The fixed header format for messages was inflexible, as new headers could not be added easily. The use of manual marshalling and unmarshalling also meant that adding new message types was difficult.
The architecture of the previous chapter only used a single transport protocol and connection type. It was not designed with the possibility of using alternatives. As already identified different protocols are likely to be required for different network configurations. The fixed network protocol meant it was impossible to implement such alternatives. This protocol also did not implement adaptive bandwidth usage or time delay management. The fixed header format would need to have been changed to implement these.

The use of access control of particular channels, although usable, was confusing and mixed two paradigms, one being the use of channels for communication, and the other the access to resources. By combining the two into a single ‘Access Controlled’ channel, the two were tied together and could only be used in one way. For multiple peers to use the same resource for access control, the second peer needed to specify that its access control was tied to that of the other. A more flexible and intuitive approach was to introduce tokens that represent resources; a token can only be owned by one peer at a time. The ownership of tokens can then be used as an indication of ownership of resources by multiple peers.

The structured protocol of messages was very restrictive, due to the fact that only certain headers could be included and that these had to always appear no matter whether they were relevant or not. This technique of fixed headers is used by low level protocols such as IP, TCP and UDP where efficiency and speed are the main criteria. An alternative is the use of named headers as used by the MIME (Borenstein and Freed, 1993) and HTTP (Gettys et al., 1997) protocols. The use of names means that the type and order of headers can be varied for different types of messages. In addition, extra headers can be added at a later date without breaking previous implementations.

The conversion to and from a byte stream of messages required manual conversion of each header. This was prone to errors and time consuming to produce. The serialising and de-serialising of data can be performed automatically by certain systems, such as Java, CORBA or RMI. To make new applications easier to write and less error prone this process should be as automated as possible.

Additionally within the router there was no clear division between router commands and messages broadcast to peers. The router interpreted commands addressed to it as local commands that were processed and replied to, while other messages were sent to their respective peers. Although this was functional, it resulted in a confusing programming interface.
7.1 Revised Framework

To address the issues outlined above, the framework was revised to implement the missing requirements and provide a clearer programming interface. The revised architecture was first described in (Dalton, 2001). The architecture was still based on the architecture of Message Oriented Middleware (MOM) (Orfali et al., 1996b; Shoffner, 1998), but new layers and concepts were added to the previous model.

The revised architecture was influenced by Sun’s Java Shared Data Toolkit (JSDT), and IBM’s Shared Data Objects (SDO). The JSDT initially looked as if it could provide many of the identified requirements of web telerobotics. However, despite being a Sun Java product, JSDT does not satisfy the web enabled requirement as Applets must be signed. In addition because the JSDT libraries take up 700 KBytes it adds significant size to the downloaded Applet. The revised architecture was required to work as an unsigned Applet, and add minimal overhead to the size of the download. Both toolkits were also Java only implementations and therefore do not satisfy the platform and language independence requirement. Both also did not implement stream separation, message prioritisation, adaptive bandwidth usage, or time delay management.

7.1.1 Contexts

Shared contexts were introduced to provide addressing, access control, and resource control. They replaced both peer/router commands and access controlled channels. In the revised architecture peers perform operations on contexts, and contexts only. There are no router commands. Access controlled channels are replaced by a specific type of context - tokens. Tokens can be used to represent resources that cannot be shared and can be owned by only one peer at a time.

Different types of contexts are used to represent types of shared entities. Contexts can be managed by a manager specified at their creation; this enables access control. The manager is consulted to accept or reject all requests to the context. All contexts support at least the following actions:

- join
- leave
- get list of joined peers

To join a context a peer must have joined its parent first. Once a peer has joined a context it can then perform other operations on that context, and will be notified when the state of the context changes (a peer joining for example). Each context maintains a list of peers that are currently joined. There are three main types of context, these are Domain, Token, and Channel.
Domains are like directories and are place holders for a group of related contexts. Contexts are always created within a domain, the default being the root domain. Management of a domain provides a default behaviour for all its children as a peer must join the parent domain first. Each domain maintains a list of child contexts that exist within it. Domains support the following operations:

- create child context
- delete child context
- get list of child contexts
- lookup child context

Tokens are used for resource management and can be grabbed and released by peers. A token can be owned by only one peer at a time. Whether a peer can grab a token is determined by the token’s manager within the MOM. If token ownership changes, all other joined peers will be notified. Tokens replace the access controlled channels of the previous chapter. Tokens support the following operations:

- grab
- release
- get owner
- give to

Channels are used by peers to exchange messages. Any message sent to a channel is forwarded to any peers it is addressed to, or broadcast if the address field is missing. Each message sent by a peer has a unique ID, which can be used by other peers to reply to it. This enables channels to support both the synchronous and asynchronous communication requirements. Channels support the following operations:

- send message
- send message and wait for reply
- send reply

An example configuration for a group of contexts is shown in figure 7.1. The figure shows the root domain, which contains two sub domains one called Lego and one called robot and a chat channel. The two domains then contain channels and tokens specific to their application. There is no limit to the number of channels or tokens (or domains) contained within one domain, the only requirement being that they have different names. This tree structure allows applications to create their own branch specific to their needs. It also means contexts can be addressed using URLs where the hostname is the name of the MOM, and the path name is the set of contexts that need to be traversed from the root domain. For example, the robot channel might have a URL such as `moms://telerobot.mech.uwa.edu.au/robot/robot.channel`. This provides quite an elegant solution to the addressing requirement.
7.1 Revised Framework

Figure 7.1: Contexts are organised in a tree structure in much the same way that directories and files are.

The contexts are managed within the MOM server, but each peer also maintains a proxy copy of each context. All operations on a context are performed indirectly via these proxy contexts. Any change in a context within the MOM server is relayed to all peers that have joined that context, so that the proxy object maintained by the peer is always up to date (synchronisation). The contexts maintained by the MOM are known as master contexts.

7.1.2 Transceivers

To allow different protocols (protocol independence), the concept of transceivers was introduced to the framework. A transceiver is an interface that provides the minimal services expected of a connection to an external process. Transceivers send and receive objects with higher application layers. Transceivers must convert the objects to and from a suitable form for transport. All transceivers must implement the same interface which is exposed to higher layers in the application. Different transceivers can therefore be used within the framework without needing to modify any higher layers. Transceivers are described in detail in section 7.4.
7.1.3 Parcels

The lack of flexibility caused by the use of fixed headers was overcome by introducing a flexible structure known as a parcel. Parcels consist of a list of keys and values; these key value pairs are termed slots. The keys are strings, but their associated values may be any object type. Parcels are the objects that are exchanged between contexts and transceivers. All context commands, events, replies and errors are passed around the MOM system as parcels. When an operation is performed on a context, this is translated into a parcel, which is in turn translated into network format by the transceiver. The reverse operation is performed when data is received over the network. In practice this translates to the named header approach, although it is actually more powerful as data within slots can contain arbitrarily complex data structures. The only requirement on these data structures is that the transceiver is capable of serialising and deserialising them.

7.2 The MOM Architecture

As in the previous architecture the MOM is the central part of the system which all peers connect to. The MOM is split into layers in with a context and communication layer. The connection layer listens for new connections, and then exchanges parcels with the connected peer. The context layer manages the list of shared contexts and their state. The connection layer must handle connections from multiple concurrent peers which introduces issues such as synchronisation and fairness in serving multiple connections.

7.2.1 Communication Layer

The communication layer plays two roles. Firstly, it accepts new connections from peers, and then notifies the MOM with the new connection it must also detect when these connections have been closed (session management). Secondly, it is responsible for sending and receiving parcels over the network between each of the connected peers. Peers may have different connection speeds. The communication layer must be able to buffer or discard parcels for delivery to slower peers (adaptive bandwidth usage). The communication layer can communicate with different protocols concurrently (protocol independence). The connection protocols to use are predefined in the MOM server’s configuration file.
Connection Establishment

For a connection to be established with the MOM, it must provide a published entry point that clients can access to initiate a connection. For example for TCP/IP socket connections, the TCP stack must be told to listen on a predetermined port, or for RMI the rmiregistry listens on a predefined port and provides a naming service to look up named objects.

To perform this in a flexible and protocol independent way, all connection protocols implement a listener interface. For each protocol in the server configuration an appropriate listener is created and started when the MOM is initialised. A listener accepts new connections, sets them up, and then informs the MOM that a new connection has been created. The connection is then passed to the MOM for all further transactions and the listener returns to the job of accepting new connections. The classes and interfaces that make up the listener interfacrer are shown in figure 7.2 on the next page. Each listener runs in its own thread waiting for a new connection; new connections are then passed to the NewConnectionHandler, which completes the connection setup.

User Authentication

Once a connection is established, the peer is authenticated to establish his/her rights within the context hierarchy. This must happen before a peer can join any contexts. Authentication consists of validating a username and password combination; if this fails the reconnection procedure must be repeated. Once a peer is authenticated they are associated with a username, nickname, user level and group ID. This set of credentials can then be used by context managers to accept or reject requests from the peer.

The module used to authenticate users is specified in a configuration file when the MOM is started up. The file gives the name of a class that implements the authenticate interface. The authenticate interface provides a number of functions:

- Check name and password pair
- Get group ID of a user
- Get users level
- Get full name of user

These functions are used by the MOM and context managers to check permissions for peer requests. For example, a domain may be setup so that only a certain group of users may create new contexts, in which case the domain manager will check with the authentication module and reject
Figure 7.2: UML Model showing the Listener interface and two implementations: an RMI and a socket implementation. Listeners call a NewConnectionHandler whenever a new connection is accepted. Higher layers configure the listener with the correct NewConnectionHandler to use.

any create request from peers that are not members of the correct group. Alternatively the user level might be used to establish if a peer can grab the token from the current owner.

7.2.2 Context Layer

When the MOM first starts up, the list of contexts is empty except for the root domain. As peers connect, create, delete, join and leave the context tree is updated and joined peers are notified.

The MOM receives requests to perform operations on contexts via the communication layer. These are in the form of parcels. One of the slots of the parcel indicates which context the operation applies to; the MOM examines this and forwards it to the appropriate context. The context examines other slots of the parcel to check the type of request and who sent it. It then checks if the operation
is allowed by the manager, and if so executes it. The execution of this operation may result in
a change to the context, in which case new parcels containing the new state will be created and
forwarded to all joined peers. Finally a new parcel containing the result of the request is created
and sent to the communication layer to be returned to the original peer.

For example, if a peer performs a grab on a token context, the MOM receives this as a parcel,
examines it and forwards it to the master token context. The token context checks who the current
owner is, and sees if the new peer is allowed to take it. If this is successful, then all peers joined
to the token are notified of the new owner. Finally, the result of the decision is returned to the
requesting peer as a reply parcel.

When the MOM is first started, a configuration file may specify a manager that restricts creation
of particular contexts within the root to certain peers. For instance, only the peer that controls the
robot would be allowed to create the top level robot domain. When a context is created, the peer
that requests the creation can also specify a manager; this can be used to manage all operations
on the context, from creation of new contexts to whether a peer can even join the context. A
peer must always join a context’s parent first, so membership of a group of contexts can easily
be managed by a single domain parent. This allows increasingly fine grained access control for
each step down the context tree. Management is particularly important for token contexts, as the
manager is used to decide whether to grant grab token requests. The types of managers a peer can
specify are currently limited to the available implementations in locally held byte compiled Java
files; this could be extended so that the peer supplied the byte code for the manager with the create
request.

### 7.2.3 Parcels

All exchanges between the context and communication layers of the MOM use parcels. The parcel
class is designed to be reusable in other software systems so contains no information specific to
the MOM. It provides a small set of functions for getting and setting data from slots within the
parcel as shown in figure 7.3. Slots are addressed by their name and can be set read only if
required. The names and types of data within each slot is completely flexible.

For the MOM system this flexibility is restricted somewhat as certain slots are always required so
that the system knows how to process the parcel. Therefore a second more specific parcel type
and an associated factory are used to manipulate all parcels within the MOM system. The MOM
parcel class defines extra methods that set and get certain slots that are used within the system.
The factory is used for the creation of Parcels, and ensures that slots such as the timestamp, and ID are set to correct values for each new parcel.

Some commonly used parcel slots include: destination address, sender, timestamp, priority, family, name, ID, reference ID, context name and payload. The destination address slot can contain a list of peers that a channel message is directed to; if the slot is missing then the message will be broadcast to all peers. The sender slot identifies the creator of the parcel. Timestamps establish time delay by indicating when a parcel was created; the parcel factory always initialises this slot with the current time. The priority of a parcel is an integer value expressing the urgency with which...
the parcel should be delivered; higher values are given higher priority. The family and name slots indicates the class that the parcel belongs to; the framework handles different classes of parcels in different ways. The ID and reference ID are used for identifying parcels; a reply to a previous parcel would include the original parcel ID as the reference ID. The context URL slot identifies the name of the context within the context hierarchy that the parcel is destined for. Finally the payload of the parcel is used to contain the contents of a channel message. This may contain any object, although in practice this must be of a type expected by the receiving peer. Higher layers can use their own data types by inserting them as binary into the payload slot. Binary data is passed through the lower layers of the MOM system unchanged. The higher level layers are then responsible for serialising and deserialising this binary data. This mechanism allows higher layers to use their own data protocol with no change required in the MOM system.

7.2.4 Queues

The MOM must handle multiple concurrent peers, all connected using different network speeds, while minimising the delay in sending data to each of these peers. If a single peer is connected over a very slow network, sending to this peer should not adversely affect sending to others. If a peer stops receiving data without closing its connection, the MOM should not block once the network buffer to that peer is full.

To avoid these problems each peer has an outgoing queue in the MOM server. Contexts place parcels onto these queues, while worker threads take them off and send them over the peer’s transceiver. If the queue becomes full, then the MOM assumes that the peer has stopped receiving data due to a problem, and automatically disconnects it. This is important as otherwise the context thread would block trying to put parcels onto the queue.

This issue of blocking threads proved to be a significant design issue in the MOM server, and getting it right proved time consuming. Some limitations of the Java library such as the ambiguous timeout semantics (Schmidt and Jain, 1997) and the lack of the normal socket select call meant that more established C++ solutions could not be directly mapped to Java. However, once low level network code is working reliably there should be no need to modify it. This provides a strong motivation for the use of a framework so that the process does not have to be repeated for each new networked application.
7.3 Peer Architecture

The architecture layers of a peer are shown in figure 7.5. The bottom two layers (context and communication) are similar to the MOM server, in that the communication layer sends and receives parcels between the context layer and the network. The context layer maintains the tree of contexts, but these are proxy contexts, and not the master contexts that are maintained within the MOM.

Above the context layer is the client layer; clients encapsulate an application that comprises of one or more contexts. Clients communicate by joining contexts and then sending or listening to events through these contexts. Clients are used to provide a logical interface that is closer to the underlying application it represents. For example, a camera that can be moved might use a control token, a command channel and an image channel. A camera client can combine all these contexts into a single camera interface that provides methods that relate directly to using the camera.
The top layer is the *presentation layer* which presents and receives data from the user. Depending on the underlying application this may communicate with an client, or may talk directly to the context layer.

![Diagram of Peer Architecture](image)

**Figure 7.5:** The layers of the MOM architecture for a robot client, shown using UML package notation. The presentation layer provides the user interface. The middle layers maintain the logical model of the remote system. The bottom transceiver layer sends and receives objects over the network. Communication between the different layers uses parcels.

### 7.3.1 Communication Layer

The communication layer handles connection and disconnection between peers and the MOM. While connected, the communication layer sends and receives parcels between the MOM and the context layer. Sending and receiving is performed by *object transceivers*, implementations of which may use any network protocol such as RMI, XML over TCP/IP, or CORBA. Transceivers are connected to the root domain, which routes parcels to and from all contexts.
To initiate a connection, the communication layer must be told which connection protocol to use, and where to find the MOM server. This information is contained within a URL. All connection protocols use the same interface to initiate a connection and can therefore be easily interchanged, as with the listener interface used in the server.

7.3.2 Context Layer

The context layer of a peer consists of a subset of the context tree held within the MOM. The contexts are proxy contexts, that defer all but the simplest requests to the MOM. The types of contexts and their relationships are shown in figure 7.6. As with the MOM there are the three main types: domain, token, channel, and also a special domain - the root domain. All contexts are derived from a common basic context. Proxy contexts maintain some information about their state. All contexts maintain a list of peers that are joined; the domain context maintains a list of child contexts; and the token context maintains the current owner of the token. These states are updated by events received as parcels from the MOM.

Higher layers use proxy contexts in two ways, either by calling context functions, or by registering to be notified of context events. The functions (such as join or leave) that can be performed on a context have already been discussed in section 7.1.1. Functions that read a context property are execute locally. Functions that change the state of the context are passed to the MOM to handle. State changing functions include: joining a context, creating a child context in a domain, requesting a token, or sending a message over a channel. Functions can have both synchronous and asynchronous variants. Synchronous functions block until a reply has been received from the MOM. Asynchronous functions return immediately, their results are made known via the event listener interfaces. Some of the events generated include:

- peer joined/left (all contexts)
- reply received (all contexts)
- child context created/deleted (domain)
- logged in/out (domain root)
- message (channel)
- token grabbed/released (token)

These various events can be received by any part of an application that registers itself with the context to receive the event. This subscription to events interface makes extensive use of the Publisher Subscriber pattern (Buschmann et al., 1996).
The domain root is a special type of context as it provides the entry point to the context layer for both the communication and higher layers. Higher layers start a connection with the MOM via the connect method of the domain root; this in turn creates an appropriate network layer for the specified network protocol. The communication layer is configured to forward all received parcels to the domain root which then dispatches them to the appropriate context. Once successfully connected the domain root’s login function must be called. The login function creates a login parcel which is sent to the MOM. When a new peer logs into the MOM, the MOM creates a series of context events to update the peer’s context tree to match that held by the MOM. With the new correctly mirrored set of contexts, the peer can then join, create, send messages, grab tokens etc.
For example, consider a scenario where the MOM context tree contains a single channel, which three peers have joined. If a new peer logs into the system, after authentication of the user, the MOM would send a child context created event for the channel. On receiving this event the peer would create a local proxy of the channel. If the peer then chooses to join this context, the MOM will send three peer joined events back to the context, one for each of the three peers that had already joined. At this point the peer will have an accurate mirror of the domain root, and the contained channel with an associated list of joined peers.

7.3.3 Client Layer

The client layer is not strictly part of the core framework, as its implementation is different for each application. However, the client concept can exist across multiple applications. Clients are used to provide a logical interface that is closer to the underlying application it represents. An application refers to some device or space that a group of contexts is being used to represent. For example, a robot might be represented by a control token, a command channel, and a position feedback channel. The robot client would provide a robot interface to higher layers, which might contain functions such as move, or get current position. To implement these functions the client would subscribe to all the robot contexts and implement a mapping between each function call and the underlying contexts.

7.3.4 Presentation Layer

The presentation layer provides the user interface in peers that need one. Peers that require no user interface do not need a presentation layer. Many different presentation layers could use the same set of underlying contexts or clients. This decoupling of the user interface and underlying networking and control structures means that many different user interfaces can be created for different types of users, or hardware platforms, while all using the same underlying control software.

7.4 Communication

Communication between peers and the MOM server is performed by transceiver objects on both sides of the connection. Although the initial connection phase is different for the MOM and peers, once a connection is established the transceiver objects used are identical. Transceivers
perform in two modes concurrently, *Upstream* and *Downstream*; this is shown by the upstream and downstream interfaces in figure 7.7 on the following page. In downstream mode a transceiver receives objects via its putqDownstream method, and sends it downstream; in upstream mode it performs the reverse (via the putqUpstream method). This is similar to the *Sender Receiver* design pattern presented in Buschmann’s Design Patterns book (Buschmann et al., 1996).

If the downstream side of a transceiver cannot receive objects (a network interface for example), then the transceiver must translate to and from a suitable transport format before sending/receiving over the interface. Using translation, transceivers may be used to connect across interfaces such as process boundaries, or across a network.

A number of different transceivers were implemented to connect between peers and the MOM, the difference in implementations, being the protocol that was used to transfer data. The formats included:

- XML
- Java Object Serialisation
- Java Remote Method Invocation (RMI)

The first two techniques used variations on a *Socket Transceiver*, while the last used a *RMI Transceiver*. These transceivers were used to connect across a network, although could also have been used to connect across processes within a single operating system.

### 7.4.1 Socket Transceiver

The socket transceiver formed the basis of two implementations of transceivers. These were the XML and Java Serialisation transceivers mentioned above. These two implementations differ only in how the objects are translated to and from a stream format.

The socket transceiver architecture is shown in figure 7.8. The transceiver contains a reference to a socket and its input and output streams. These streams are the downstream side of the transceiver. The transceiver contains *sender* and *receiver* objects that run in their own threads. The sender periodically writes outgoing data to the socket output stream (this buffers data to avoid sending tinygrams, as discussed in chapter 5 on page 77). The receiver constantly monitors the socket’s input stream. The upstream side of the transceiver consists of a reference to an upstream object that all upstream objects are forwarded to. Between the upstream and downstream parts of the transceiver is the translator, which operates on both upstream and downstream objects. The translator converts between the socket’s stream format, and the objects that are passed to and from
Figure 7.7: The transceiver interface is a generalisation of the upstream and downstream interfaces. Although these interfaces are identical in functionality, the use of different names makes direction of message flow easier to follow within the source code.

upstream. Different network protocols can be implemented by using different translators within a socket transceiver.
Figure 7.8: The Socket Transceiver contains a sender and receiver object that run in their own threads. Errors are propagated to other parts of the system via the manager. The translator object handles the serialising and deserialising of objects to a stream form.

Translators

Two type of translators were implemented, one using an XML/MIME format and the other using Java’s in built Serialisation. The architecture of both these types is shown in figure 7.10 on page 137. The Java translator is extremely simple as it uses Java’s runtime support for serialisation of objects, whereas the XML based translator requires a large hierarchy of helper classes.
Figure 7.9: Interaction between the socket transceiver and the translator. Objects are converted to a binary format by the translator and then sent over the socket by the transceiver.

Although the Java translator is simple, it has the disadvantage of using a protocol that only other Java applications would be able to understand (thus breaking the Platform and Language Independence requirement). The XML translator produces a format that is easily read by applications written in any language on any platform, but this comes at the cost of additional complexity.

MIME Translator

This translator uses an XML/MIME combination as the stream protocol to send and receive objects, and is therefore known as the MIME translator. XML provides the flexibility to send complex data structures within nested tags. XML is becoming increasingly popular as a data interchange language for the Internet. As a result there are both free and commercial parsers available for most programming languages and platforms.

A disadvantage of XML is that it is designed for the exchange of textual data only. These text documents can be large compared to optimised network protocols such as CORBA - XML can be larger by a ratio of 10:1. Binary data must be encoded before being included. Data also cannot include the reserved characters ‘<, /, >’. These special characters can be encoded or the containing sequence escaped using a CDATA section.

For large amounts of binary data encoding/decoding is slow and increases the size of the data further; an alternative solution is to use the MIME multipart/related content type (Levinson, 1998). Binary data is included as a separate part of a message. This was the technique implemented for the MIME translator. An example of a serialised MOM parcel is shown in figure 7.11 on page 138. The first part of the message indicates that it is the multipart/related type and that parts of the message are delimited by the boundary string. The next section is the object serialised as XML. Each part of an object is translated into an XML tag, whose name
Figure 7.10: UML diagram of the two types of translators used by the socket transceiver. The Java translator is very simple, as the serialisation and deserialisation of Java objects is supported within the language. The XML translator requires a hierarchy of helper classes to perform its translation.

is the variable name within the enclosing object with a type attribute that indicates its type. For example, a field named ‘username’ of type string with value ‘jack’, would be written as `<username type='string'>jack</string>`. The type field is important for polymorphic types to indicate which derived type the value is. For example the set of parcel slots is stored in a Java hashtable which maintains a list of keys and values; the type of these elements can be
any Java object. To be able to recreate the hashtable from the XML stream, the exact type of all the contained objects needs to be known. The char type is special, as this is an array of characters - the binary type referred to earlier. The binary tag references an external entity - the cid:body which is included as the final part of the message in its raw binary format.

```
Content-Type: multipart/related; boundary=xxxxBOUNDARYxxxx

xxxxBOUNDARYxxxx
Content-Type: text/xml
Content-ID: object

<?xml version="1.0" ?>
<parcel type="parcel">
   <properties type="hashtable">
      <key type="string">to</key>
      <value type="list">
         <value type="string">barney</value>
         <value type="string">mark</value>
         <value type="peer">robert</value>
      </value>
      <key type="string">context</key>
      <value type="context">telerobot</value>
      <key type="string">body</key>
      <value type="chars">cid:body</value>
      <key type="string">priority</key>
      <value type="integer">10</value>
   </properties>
</parcel>

xxxxBOUNDARYxxxx
Content-Type: application/unknown
Content-ID: cid:body
Content-Length: 50

%<a[$^&()@%^$^&$%@^$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$%^$^&$

```

**Figure 7.11:** An example of a parcel serialised as XML and binary using the multipart/related MIME type. The first part of the message indicates that it is the multipart/related type. The next section is the object serialised as XML. Part of the XML references an external entity - the cid:body which is included as the final part of the message in its raw binary format.

The MIME translator contains an **XMLWriter** object that maintains a database of registered object types, and an associated object that will translate them to XML. Some of the types of writer objects are shown at the top of figure 7.10 on the preceding page. A XML writer must implement
the function `toXML` which takes the object as an argument and returns a complete representation of itself and any contained objects as an XML string.

To translate an object into XML, the translator calls the XML Writer for the top level object, which in turn calls writers for any enclosed types. Any contained binary types are handled by the above mentioned external reference technique. The final result is shown in figure 7.11.

Recreating an object from its XML representation is more complex than generating it, as the XML must be parsed into tokens. However, due to the design of XML this can be performed by a standard XML parser with little additional work. This does mean that a XML parser must be included in the code. As the target environment included Java Applets, the size of any required code needed to be kept to a minimum. The smallest parser available was the SAX event based parser, which was also open source so had no licensing limitations. To recreate an object, the SAX parser is invoked with the raw XML. It then calls a registered document handler for every XML tag that it reads. This was the `XMLReader` class shown in figure 7.10. The `XMLReader` class maintains a list of registered readers in much the same way as the writer. As tags are read in, the `XMLReader` calls the appropriate reader to fill out the new instance with the enclosed data. The final product being an object that mirrors the MIME stream that was passed in.

**Java Object Serialisation Translator**

The other translator used Java’s object serialisation to convert objects to and from a stream format. By comparison to the MIME translator this class is simple, as it uses Java’s `readObject` and `writeObject` functions that work automatically for any Java class. These functions use Java’s introspection to automatically serialise and deserialise objects to a binary stream. One caveat of using this technique is that a class serial number is used, and if the sender and receiver are using classes with different serial numbers, then automatic deserialising will not work. The simple solution is to make sure the serial number is constant; however if a class changes then the serial number will need to change. However, obviously the main problem with this approach is that it introduces a Java only protocol to the system, this is discussed more in section 7.4.4.

**7.4.2 RMI Transceiver**

The RMI transceiver is an alternative to the socket transceiver which delegates all network communication to Java’s RMI. Instead of using a socket to send and receive streams, objects are passed
directly between peer and MOM as arguments to remote procedure calls. As RMI is a Java only solution both sides of the connection must use Java code in order to communicate.

![Diagram of RMI Transceiver object hierarchy]

Figure 7.12: The RMI Transceiver object hierarchy. All objects that handle remote RMI function calls must implement an interface that is ‘Remote’.

Figure 7.12 shows the class diagram for the RMI transceiver and its helper classes. The RMI transceiver implements both the normal transceiver interface, but also a remote one. It must also extend the UnicastRemoteObject object so that it can act as a RMI server as well as a client. The RMI MOM Connection class extends the transceiver to include the connect/disconnect functions. The RMI Listener also implements two interfaces, the standard listener interface and a RMI version; it also extends the UnicastRemoteObject so that it can act as a server.

To establish a RMI transceiver connection between a peer and the MOM, the peer uses the RMI naming service to lookup and receive a stub for the Listener object. The peer then calls the Lis-
RMI provides a higher level interface to distributed programming than using sockets directly. The low level creation and exchange of data over sockets is hidden from the programmer. In fact, the underlying protocol can be changed without any modification to the client code. However, for this application using RMI created more problems than it solved. Using RMI from an Applet, RMI callbacks require a socket to be opened in listen mode which is forbidden in the standard security model; added to this the Microsoft version of Java included with Internet Explorer does not implement RMI at all. RMI is supposed to be able to tunnel over HTTP when it encounters firewall problems, however the programmer has no control over when this happens, and due to problems in the Netscape Java virtual machine this did not work properly either. A more fundamental problem is that RMI is a technique for synchronous remote procedure execution, whereas the MOM is an asynchronous message based system. To make RMI behave asynchronously extra threads must be included in the client and server so that calls can return immediately; this adds another layer on top of the RMI layer, which is in turn layered on top of the underlying network. RMI is providing a similar level of abstraction to the MOM, and to use it as the low level protocol within the MOM is probably misplaced.
7.4.3 Other Transceivers

To develop a new transceiver a class only need to implement the transceiver interface. There are therefore many other possible implementations other than the two mentioned so far. The actual transceivers used with a system can be easily varied without needing to change any of the higher level code. These may use the Socket transceiver of section 7.4.1 but merely provide a different translator, or they might use a whole new underlying communication technique. The following sections discuss some possible types of transceivers.

CORBA Transceiver

CORBA provides a similar RPC interface to RMI, but at the same time provides cross platform and language inter-operability. A CORBA based transceiver could therefore provide similar platform independence to the MIME transceiver while automatically handling the serialising and deserialising of objects. However, a CORBA transceiver is likely to suffer from the same synchronous call problems that the RMI transceiver did. Also CORBA support in browsers is minimal or nonexistent so there is little advantage to be gained if the target application is an Applet.

HTTP Tunnelling transceiver

As discussed in section 5.3.1 on page 85 not all clients are able to create socket connections on unprivileged ports due to firewalls. For these clients, an alternative technique would be to use HTTP tunnelling. An HTTP tunnelling transceiver would communicate with a Java Servlet on the server side to relay parcels to the MOM. The transceiver would accumulate send requests and periodically send these wrapped within an HTTP request; the URL of this request would be a CGI script, most probably a Java Servlet, that would act as a relay between the peer and MOM. This Servlet would maintain a permanent connection with the MOM and would be a proxy peer connecting via a socket transceiver for example. The Servlet would also accumulate messages sent from the MOM destined for the peer. These would be wrapped in an HTTP response to the original HTTP request. As the Servlet would have no way of pro-actively contacting the client, the tunnelling transceiver would have to periodically send requests even if it had no data to send.
7.4 Communication

Translators

Other stream protocols could also be defined for the Socket Transceiver by developing new translators. A structured binary protocol such as that described in the previous chapter could be used, but as was found before this can have problems in being inflexible to changes in data structure.

A better, highly efficient alternative for the underlying protocol would be to use the Abstract Syntax Notation ASN (Larmouth, 1999; ITU, 1997a), as used in many telecommunications applications. ASN allows complex and dynamic data structures to be defined in a language and hardware independent way. Data structures are defined using the notation, which can then be encoded using one of a number of rules such as BER or packed encoding rules (PER) (ITU, 1997b). This can be viewed as an alternative to CORBA or other RPC mechanisms, where the structure of the message to be passed is defined, instead of the remote procedure that can be called. This is much closer to the MOM implementation, where the ASN structure encoding can be seen as another way of serialising the data.

7.4.4 Comparison of Transceivers

Different transceivers will be required for different situations. A single transceiver type does not need to be used by all peers, some can be using socket transceivers, while others might be using the HTTP tunnelling transceiver, while local ones might be connected using an in process transceiver.

Of the implemented transceivers, the MIME transceiver provides the best cross platform solution, while the Java serialisation transceiver requires the least coding effort to get working. Although initially it was thought the RMI transceiver would provide the quickest and easiest solution, the difficulties of mapping the RPC based semantics onto asynchronous messaging, and the implementation problems in many browsers, meant that it is of little use.

Although promising platform independence there a number of disadvantages to the use of XML. Firstly new data types must have a defined XML translation, which needs to be registered with the MOM. Secondly, XML is verbose, and can therefore increase bandwidth requirements; this could be reduced by compressing the data before sending, at the cost of extra processing time. However, XML is fast becoming an accepted standard and there are many class libraries for parsing and generating it. It also has the advantage that it is human readable which makes it simpler to discover errors.
An alternative technique to provide a cross platform solution is for all peers to use the Java translator. This can be achieved with code written in other languages by using the Java Native interface (JNI). JNI is a native programming interface that allows code running within a Java virtual machine (VM) to interoperate with applications and libraries written in other languages such as C/C++ and Fortran. Thus by using JNI it is possible to provide a Java front end to platform specific or numerically intensive code written in a different language. If an application is converted in this way, then it can make use of Java Object serialisation transceiver. However it should be noted that using this technique will tie a system to Java only solution.

7.4.5 Adaptive Bandwidth and Stream Separation

The first implementation of this new architecture used a single transceiver per peer to send and receive all objects. This worked, but did not provide adaptive bandwidth usage or stream separation. A TCP socket provides a ordered and guaranteed stream service, which can introduce delays if time critical data is queued behind other data in the stream sequence.

To address stream separation, peer connections were expanded to use two transceivers. Based on a bulk indicator slot, each parcel could then be routed down a bulk or control transceiver. High volume data such as images are sent using the bulk transceiver, while small parcels of control and state information were sent over the control transceiver. To what extent control information is able to overtake bulk information, depends on the underlying TCP/IP stack and operating system. This issue is an accepted limitation of the TCP protocol, and is the reason for the development of the SCTP (Stewart et al., 2000) protocol. An SCTP based transceiver would achieve the same objectives but allow more than two logical streams to be used.

Adaptive bandwidth use also needed to be addressed within the MOM. Some clients on slow connections might not be able to receive data at the rate that other peers are producing it. This manifests itself as a blocking write in the MOM server once the sending socket buffers are full. Without further action the associated queue becomes full, and the MOM thread blocks causing the server to hang. Solutions to this include: have a large queue and hope it does not get full, disconnect the client, or not deliver all data to the client. The last option was the one implemented. Parcels are guaranteed to be delivered unless they are marked as discardable. Discardable parcels will normally be delivered, but when a client is lagging behind in receiving data, they maybe discarded.

An alternative implementation might use a UDP transceiver for parcels that do not require reliable transport to reduce the latency of the system. However, as discussed in section 5.3.2 on page 87,
using UDP from within an Applet is only possible for a small number of clients due to firewall and Java implementation problems.

7.5 Source Code Example

This section gives a simple example of the framework in use. The example is a peer that subscribes to a chat channel and logs any messages sent over the channel to a file. This only uses a small subset of the framework but shows a peer connecting to the MOM, joining a context and then waiting for message events.

Figure 7.14 shows the source code for the example; it consists of a single class - ChatLog. The class has three functions. A constructor which creates a domain root and joins the context; a message callback function that is called by the framework; and the main function that is the Java entry point. To keep the code short, proper error handling is not included; errors are just logged to the console.

The constructor is called by the main function with a URL and file to write to. The URL specifies the type of connection to use (RMI, MIME socket, or Java serialised socket), and the hostname and port number to use. To create a domain root connected to a MOM server on the telerobot server, the URL might look like moms://telerobot.mech.uwa.edu.au:4444/, where moms specifies the protocol type and the rest of the URL specifies hostname and port number. After creating the domain root with the URL (line 13), the login function is then called to initiate a connection and login to the MOM (line 14). If an error occurs during this process an exception is thrown; in this example exceptions are all handled very simply by the exception handler on line 19. Once logged in, the peer obtains a reference to the chat channel (line 15), and the channel is joined (line 17). The peer then registers itself as a message event listener (line 18). The class must implement the MessageEventListener interface in order to do this (line 1).

Once the peer has setup and subscribed to the chat channel the MOM framework runs in its own thread. Messages received on the channel are sent to the logging class via the message function (line 25). The message function is called with two arguments: the source of the message event and the message itself encapsulated within a parcel. The peer then writes the time, message sender, and its contents to the log file (line 29). The contents of the message are obtained from the MOM parcel via the getArg function; this returns a pointer to an object which is cast to an array of bytes; as this is the format that is used by all peers to send messages over the chat channel.
public class ChatLog implements MessageEventListener {
    Channel chatChannel;
    DomainRoot sr;
    Peer us;
    BufferedWriter writer;

    public ChatLog(BufferedWriter writer, String momurl) {
        this.writer = writer;
        try {
            sr = new DomainRoot(momurl); // create domain root
            Peer us = sr.login("guest", ",", "observer"); // login via domain root
            chatChannel = // lookup or create the chat channel
                (Channel)sr.lookupOrCreate(new ContextURL("chat.channel"));
            chatChannel.join(); // join the chat context
            chatChannel.addMessageEventListener(this); // register for callbacks
        } catch (MOMException e) { // something went wrong on setup
            e.printStackTrace();
        }
    }

    // called by MOM framework when a message is received on channel context
    public void message(Context source, MOMParcel m) {
        try { // log the contents of the message to a file
            Date now = new Date();
            writer.write(now.getTime() + "!!" + now + "!!" + m.getSender() + "!!"
                + new String((char[])m.getArg()) + "!!");
            writer.newLine();
            writer.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main (String[] args) {
        try {
            BufferedWriter logfile =
                new BufferedWriter(new FileWriter(args, [0] true));
            ChatLog chatLog = new ChatLog(logfile args[1]);
            Integer i = new Integer(1);
            synchronized(i){i.wait();} // wait forever callbacks occur in
            // seperate thread
            logfile.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

Figure 7.14: Source code for the chat channel log peer. This peer subscribes itself to the chat channel. The message function is called by the framework every time a message is sent on the channel.
The main function is the entry point of the application, and is called first by the Java virtual machine. It calls the class constructor, and then waits indefinitely while the framework listens for new messages. A more complete application would handle errors, and provide a command line interface to cleanly shut down the application.

This is the only application specific code required to run this peer; all the other classes used are part of the framework. This shows that the framework provides an interface that requires very little additional work by a programmer. The work required in writing a new application is reduced to the presentation (and client) layers only.
Chapter 8

The Framework in Practice

This chapter shows how the final MOM framework was applied to the UWA Telerobot system. The system began operation in early 2000, and was maintained by the staff at UWA\(^1\). The first few sections discuss the telerobot software architecture which is layered on top of the MOM framework. This includes both the peers that control devices (robot and cameras), and an Applet interface that users download. Further sections present results and user feedback from operation from February to December 2000. These are compared to those obtained with the CGI system.

An unanticipated application of the framework was its use in a semi automatic camera/robot calibration program that could be run while users were logged into the system. The previous CGI system required a laborious calibration process that had to be performed with the system off-line. This is discussed in section 8.7. A final section discusses how the framework might be used for the UWA Telelabs project. This project aims to make certain laboratory experiments available over the web so that they can be accessed 24 hours a day, and provides a real application of web control techniques.

8.1 MOM Based Telerobot System

The framework was used to control the robot arm via a Java control Applet distributed using a web server. The full system consisted of the MOM, a robot server, an image server, and a Java Applet dynamically downloaded by users. Also included were a workspace model server, and a chat server which provided collaboration facilities between users. The term server is used here to refer to a MOM peer that tends to process commands, rather than initiate them, conversely a client is a peer that tends to initiate commands. From the MOMs perspective clients and servers are identical and are all viewed as peers.

\(^1\)The author left Australia in February 2000 - see section 1.3
The Applet interface system presented a single homogeneous telerobotic control interface to the user, but the underlying architecture consisted of a number of logically separate clients modules. Clients use underlying shared MOM contexts to communicate but do not expose these to higher layers. The clients provide a facade (see Gamma et al., 1994, page 185) interface to the remote device they represent. There are both robot and image client modules within the Java Applet.

Client modules, and their associated server peer are tightly coupled, as both must use the same shared contexts and use the same parcel structures to be able to communicate. The combination of a client and server is termed a subsystem. The system contains robot, image, chat, and workspace subsystems.

8.2 Robot Subsystem

The robot subsystem consisted of two parts, a robot server which is an evolution of the robot server of previous chapters, and a robot client which communicates with the robot server, via the MOM. The robot client is part of the client layer of the peer architecture and is a module that can be used as part of a peer. The robot server is a peer in its own right. Both the robot server and client connect into the MOM, and join the same shared contexts. The robot server creates these contexts when it first logs in. The contexts used are a robot control token and a robot channel, both of which are contained within a robot domain.

8.2.1 Robot Server

The robot server creates a robot domain, under which there is a robot channel and robot token. The token is created with a manager that will only allow a change of ownership if the requesting user has a higher access level, or if the token ownership has not been renewed for a certain period of time. Robot commands are received by the server over the robot channel. The server will only accept robot commands from the current owner of the robot token. Whenever the robot moves, its new state is broadcast by the server over the same channel. The state of the robot consists of the robot’s pose, the gripper position and the execution mode.

Due to time limitations the MOM peer architecture was only implemented in Java, so the previous C++ based robot server needed to be adapted to work within a Java environment. The two possibilities were to make the C++ code part of a DLL and access it via the Java Native Interface (JNI), or to have an intermediate Java application that bridged between the MOM and the C++
8.2 Robot Subsystem

server. As the C++ server had a user interface already, that could not be easily incorporated into a DLL, the second technique was used. Two sockets were used to connect a Java robot proxy server application with the real C++ robot server. One socket (the command socket) was used for execution of robot commands, and the other (the state socket) was used to provide continually updated state information. The command socket used a synchronous protocol consisting of pairs of robot commands and replies. The state socket was used by the C++ server to send asynchronous state updates using a separate thread. The thread continually monitors the robot and sends a new state message whenever it detects a change in the robot.

Robot Proxy

The robot proxy communicates with the MOM on one side and the robot server on the other. To communicate with the MOM the proxy joins the robot channel and token contexts. To communicate with the C++ robot server it connects to the C++ server’s control and state sockets. The proxy’s job is to translate between parcels sent over the robot channel and commands and states sent and received over the C++ server’s sockets.

The proxy receives commands from users relayed via the MOM over the robot channel. The proxy first checks that the command comes from the current robot token owner. If it does, it continues processing the request. The command is received as a serialised command object within a parcel. This is deserialised back into a command object, and then converted to an ASCII format that the C++ robot server understands. The proxy then waits until it receives a reply from the robot server. The reply is put into a new reply parcel and the parcel is then sent back over the robot channel to the originating peer. There are two types of commands, a script based command GBHCommand and a path based command MMoveCommand as shown in figure. The script based command stores the command as a string using the script language specified in section 3.8.3 on page 50. The path based command stores a series of poses which contain the waypoints for the robot to follow. Either of these types of command can be used. The script command tends to be used for commands typed directly by a user, while the path based commands are more easily generated from a spatial interactive such as the stick cursor of section 8.5.1.

A second thread in the proxy server constantly reads new robot states from the state socket; each new state is put into a parcel and broadcast over the robot channel. This means that while a move is being performed and the command thread is waiting for a reply, new state information will be regularly sent over the robot channel. This gives immediate feedback as to what the robot is

---

2The reference ID slot is set to the ID field of the original robot command parcel
The framework notifies the robot proxy when peers join and leave contexts.

C++ Robot Server

The C++ server was very similar to the original server used in the CGI system, the main difference being the addition of the second socket used to broadcast state information. The constant state updates were achieved by using another thread within the robot server that continually polled the robot controller asking it for its current pose, as there was no way of asking the robot to provide constant state updates autonomously. Due to the single threaded interface to the Robcomm control, both command and state threads were serviced concurrently using the same Windows messaging technique presented in section 3.11.1 on page 55.

8.2.2 Robot Clients

The robot client module provides an interface that peer applications can use to communicate with the robot. It provides two reusable interfaces, one for controlling, and one for observing. The control interface is a superset of the observer interface. The observer interface can be used to examine the robot’s current state, and to register for notification of new states. The control client extends this interface and allows a user to request/release control of the robot, and execute commands.

To use either client in an application they must be initialised with a MOM domain root. Both clients then use the domain root to lookup and join the robot domain, and robot channel. The control client also joins the robot token as it needs to use it to request and release control of the robot. To receive updates on the state of the robot, the application can register with the client to receive events asynchronously, or explicitly ask for the current state synchronously. These states can be used in conjunction with the command state (see the next paragraphs) to supervise the robot. As the state information is small, compared to video feedback for example, this information will be delivered with less delay.
Controlling the robot using the control client can be synchronous or asynchronous. In synchronous mode an execute command call will not return until the robot has finished executing it. To execute a synchronous command the following sequence of events takes place: A robot command parcel is created and sent to the MOM, which forwards it to the robot proxy. The robot proxy checks that the parcel is from the current controller and, if it is, forwards the command to the C++ server. The C++ server executes the command and creates a reply message which it sends back to the proxy server. The proxy server puts the message into a parcel and returns it via the MOM to the client. The client receives the reply to the command and returns the result to the user. The client makes use of the MOM framework’s ability to send synchronous acknowledged parcels to correctly identify the right reply parcel.

To control the robot in asynchronous mode, the robot server and client maintain a list of pending robot commands. On the client side these have a state that indicates their current status, this can be one of: **sending**, **executing**, **finished** or **error**. Whenever a new command is sent it is added to the pending list of robot commands. This enables a user to send a command without having to wait for the previous one to complete, while still being able to receive feedback on each individual command. The list of commands is **observable**, so that whenever the status of any of the commands within the list change, registered parts of the application can be notified. This notification can be used to update the command display in the user interface.

### 8.3 Image Subsystem

The image subsystem is split into a similar client-MOM-server architecture to the robot subsystem. The server controls a number of cameras which it uses to take new images. Each camera has a corresponding channel that is used by the server to send new images, and by clients to send camera commands. All camera channels are created within an enclosing domain. Image clients subscribe to the domain and any camera channels that they require.

#### 8.3.1 Image Server

The image server is designed to work with the robot server, but it can also operate in a simple image taking role on its own within the MOM system. The role of the image server, is to take new images of the robot as it moves, and send these to all peers that are watching the robot. To do this it creates its own image channels, but must also subscribe to some of the robot contexts to receive state updates. Knowing the robot state at all times means that images can be taken only when the
The Framework in Practice

robot is moving. It also means that cameras can be directed at the current gripper end point, to provide images that track the movement of the robot.

On startup the image server connects to the MOM and creates a camera domain, under which it creates a channel for each camera. The number of cameras, their names, and properties are specified in an XML configuration file. Each camera’s channel is used by clients to send commands and by the server to broadcast new images. Commands can be used to change camera settings, and to request new images. New images are broadcast over the channel as a parcel that includes the image pixels (in JPG format), its size, its calibration matrix (for those cameras that are calibrated), and its URL. The URL can be used to refer back to the image at a later date. The calibration matrix is included to enable the image to be used within a virtual or augmented reality interface or for tasks such as object recognition.

The image server uses a robot client module to subscribe to the robot contexts. This is not only used to monitor the robot state as mentioned above, but also to monitor the current controller of the robot. It was decided that the system should use similar access rules to camera settings as the CGI system, so only the robot controller is allowed to change image specifications. This is enforced by monitoring the robot token and only accepting commands from the current owner.

Once all the contexts have been created and the image server has joined (become a member of) them, it monitors the robot channel waiting for the robot to move. Whenever the robot moves the image server takes new images of the workspace (optionally using the robot’s position to centre images about the gripper endpoint). For each new image taken by a camera a parcel containing the image details is created and sent over the respective channel. This continues until the robot stops moving, when a final set of images are taken. This process is repeated every time the robot moves. An interesting effect of this is that new images are taken, even if the robot is moved from within the lab via the teach pendant, as this generates new state events on the robot channel.

Image Streaming

Initially the maximum throughput of the MOM was not known so the first implementation of the image subsystem sent only low bandwidth information about the image through the MOM. This included the URL of the image which the client could then request using normal HTTP. This meant that the messages sent through the MOM were small, and the transfer of image data was delegated to a web server. However this had the disadvantage of introducing an extra round trip delay. For a client to download an image, the details must first be received over a channel, an HTTP request
made, and finally the image returned, a total of one and a half round trips. This is compared to sending the complete image in the initial message where only a single one way trip is required.

To reduce the delay, the image details sent through the MOM were changed to include the binary image data. To minimise the impact on time critical data such as robot commands, the image parcels were marked with the *bulk* and *discardable* flags as discussed earlier in section 7.4.5 on page 144. Image parcels can therefore be discarded within the MOM if a client is not processing parcels at the rate they were being produced. As a streaming video solution this is very crude as it is does not make use of similarities between consecutive frames, nor does it allow for variations in stream qualities and data rates that can be scaled to different client connection speeds.

If a user is on a low bandwidth link then they may not want to receive all images from all cameras for each frame. As each camera is a separate channel, all a client need do is unsubscribe from a particular channel and they will receive no new images from the associated camera. To turn the camera back on again, they just subscribe to the channel. The image server listens for subscription requests and whenever a client joins a camera channel, it sends the most recent camera image to the client. This ensures that the client does not have to wait for a next image to be taken (only taken when the robot next moves).

**Core Implementation**

The image server was an evolution of the server used within the CGI system. However, as the image server did not already have a user interface, the C++ code that controlled the framegrabber was converted to a DLL that was then accessed using the Java Native Interface (JNI). The interface between the MOM and the image server was handled by the Java part of the image server, while the C++ DLL performed the image grabbing process. This meant there was no requirement for a proxy and C++ server needed by the robot subsystem.

Figure 8.1 shows the core architecture of the C++ camera DLL: a camera manager maintains a list of cameras which are referenced by name. A camera is actually an abstract base class that can represent many types of cameras. Of these the implementation used was a framegrabber based camera. Each framegrabber camera has an associated framegrabber board, and also a video switching object. Two types of framegrabbers were used in the project: the Meteor and the Meteor II. The first Meteor did not have video switching capabilities so an external video switcher was used, while with the Meteor II video switching was performed internally. Although only framegrabber based cameras were used, the camera model is designed to accommodate other types of cameras such as the now widely available webcams that attach via a parallel or USB port. Each camera
Figure 8.1: The underlying architecture used to represent cameras, their position, and the framegrabbers that may be needed to access them.
8.3 Image Subsystem

can have an associated calibration matrix, and supports a `setCentre` command to point it at a specific location in the workspace. As the lab cameras were fixed this was performed in software by moving the grabbed rectangle as close as possible to the specified workspace point. This was limited and only really had an effect when the area being grabbed was significantly smaller than the full framegrabber frame. However, cameras supporting a physical pan, tilt and zoom could support this function fully to really follow the robot as it moved within the workspace.

![Class Diagram: camera / ImageServer](C:\Users\Barney\uml\mom.mdl)

**Figure 8.2:** Java architecture of image server

The Java side of the image server also maintains a list of cameras; each of these cameras is a proxy object that calls through to an underlying C++ camera object. Cameras can have cameras listeners. A camera monitor is used to monitor events from both the camera channel, but also the camera itself. The channel is monitored for camera commands, and for peers joining and leaving. Commands are executed by first being deserialised and then *double dispatched* (Eckell, 1998) to the camera via the command’s `dDispatch` method. Whenever a camera takes a new image it
The Framework in Practice

notifies the camera monitor, which packages the image into a new parcel and sends it over the image channel to all joined peers. The code to perform this is shown in figure 8.3. The parcel is marked as being bulk and discardable before being sent.

```java
public void newImage(Object source, ImageData image) {
    try {
        if(image.pixels == null) {
            try {
                // Copy the image from file into the image.pixels
                ByteArrayOutputStream bao = new ByteArrayOutputStream();
                CopyStream.copy(image.filename, bao);
                image.pixels = bao.toByteArray();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        // Remember this image as the last image (used when a new peer joins)
        lastImage = image;
        // Create a new message set various flags and inset the image
        MOMParcel message = channel.getParcelFactory().createMessage();
        message.setBulk(true);
        message.setDiscardable(true);
        message.setArgSerialized(image);
        channel.sendXMessage(message); // Send it over the channel to everyone except ourselves
    } catch (MOMIOException e) {
        e.printStackTrace();
    }
}
```

Figure 8.3: Code for sending a new image to a peer. The image data is first loaded from disk, and then put into a parcel. The parcel is created using the channel’s parcel factory, which creates the parcel and also sets some default values, such as the context and timestamp. The image data is serialised and inserted into the packet. Finally the packet is sent over the channel to all peers except the image server.

8.3.2 Image Client

The image client module provides an interface to controlling cameras, that hides the underlying camera domains and MOM connection. It implements the same interface as the one used within the image server, but instead of deferring commands to the C++ DLL, they are serialised and sent over the camera’s channel. These commands were implemented to operate asynchronously to avoid hanging the calling thread while the camera operation was executed. As two years of telerobot operation had shown that camera operations were very reliable it was decided that the
benefit of asynchronous operation outweighed the need to check return codes\(^3\). Any errors that did occur with a camera were received as messages, and relayed to the user via a general purpose console.

Figure 8.4 shows some client camera code to change the settings of remote camera. First a new parcel and new settings command object are created. The command is then inserted into the parcel as a serialised object and is then sent over the camera channel. Any network errors are caught by the exception handler and logged to the console. One command that is not relayed to the remote camera is turning it on or off, as the real cameras are always on. If a client wants to turn a camera off, they simply unsubscribe from the camera’s channel. To receive images a part of the application can register as an image listener with the camera module. Listeners are notified whenever a new image is received.

```java
public void setSettings(CameraSettings newSettings)
{
    try
        // create a new message
        MOMPacket message = channel.getParcelFactory().createMessage();
        CameraCommand command = // create new command object
            new SettingsCommand(newSettings, SettingsCommand.SET);
        // serialise command object
        message.setArgSerialized(command);
        channel.sendMessage(message); // send message over channel
    } catch (MOMIOException ex) { // catch any network errors
        TheConsole.println
            ("Problem changing camera settings: \(" + ex.getMessage());
    }
}
```

**Figure 8.4:** Client code to change a camera’s settings. A new parcel is created using the channel’s parcel factory; this is filled with a new settings command and then broadcast over the channel. The broadcast means that other users can see that a camera command has been sent.

### 8.4 Peer Collaboration

The image and robot subsystems described so far, although very different to their original CGI implementation, nominally provide the same services - although in a much improved and extended way. However they do not use the framework features that enable peers to communicate and collaborate with each other. This section describes a couple of collaboration tools that were implemented over the framework. They are very simple, but show how the collaboration aspect of the framework can be used to provide services that would have been extremely difficult under the CGI system. The two services described are a chat tool and a workspace modelling tool.

\(^3\)This is apart from network errors which are propagated as exceptions regardless
8.4.1 Chat

The chat tool allows users to exchange messages with each other. Each peer has a chat interface which consists of three panes, one to type new messages, one to see received messages, and one to list other members of the chat channel. Each peer subscribes to a chat channel, and sends and receives messages over it. Compared to well developed chat clients such as IRC this is very basic, but provides a basic forum for users to exchange ideas and instructions, with very little additional code required. No chat server is required, as the MOM automatically handles the basic chat requirement - forwarding messages to all peers subscribed to the channel. Figure 8.5 shows most of the class that implements the chat tool. The `setChannel` function is used to set up the MOM channel; the class registers as a listener for message and peer events, and joins the channel. Because the class has registered itself with the MOM framework, the next three functions are called by the framework whenever events happen on the channel. If a peer has joined or left, the list of peers is updated, if a message is received it is displayed to the user. The final function (`actionPerformed`) is called whenever the user presses the return key. Any text that they have typed is put into a parcel and sent over the channel. All that is required to complete the class is some functions that setup and manage the user interface.

To provide users with help, a hint server peer also subscribes to the chat channel. This broadcasts one line hints read from a file over the channel at regular intervals. The hint server was designed as a way of providing incremental help to users, while they were using the robot, without requiring them to spend time reading help files and a tutorial. The hints are provided as chat messages that appear in the chat console as messages just like those from any other user. The addition of the hint server required no change in code to any of the peers, or MOM. In fact it was developed (in a matter of hours) and started without requiring a restart of the MOM server.

8.4.2 Workspace Modelling

The workspace modelling tool was introduced to allow multiple users (or autonomous agents) to collaborate on a model of the workspace that could then be used for higher level commands to the robot (see section 15 for a discussion on the subject of workspace modelling). The subsystem consisted of a workspace server, that creates and subscribes to a workspace channel. The server maintains the workspace model as a list of objects, which is mirrored by each peer. Peers can add or remove objects by sending workspace commands over the channel. Changes to the model are then broadcast to all peers. When a new peer joins the workspace channel they are sent a series of workspace updates to bring their model up to date with that held by the server.
Figure 8.5: Class used to implement the chat tool. These functions are all that is required to implement sending and receiving of messages, as well as receiving notification of other peers joining and leaving the channel.

As all peers share this model it can then be used in instructions to the robot, or cameras, or to other peers. Section 8.5.1 discusses how the workspace model was used in the client user interface.
8.5 Client Applet

The client Applet combines all the client modules mentioned so far into a single user interface, that enables the user to control and observe the robot, chat with other users, read hints, and model the workspace. The interface (shown in figure 8.6) has four main parts; a robot command panel, an image panel, a chat panel, and a console area. The robot and chat panels relate directly to an underlying client module. The image panel is more complex and combines information from the robot and camera modules to provide an augmented reality interface (see section 8.5.1) that can be used for both observing and controlling the robot. The camera and robot panels are discussed in more detail in the following sections.

![The Applet interface used to move the robot. The interface is divided into four parts. The top left corner refers to the robot, and the top right corner shows images taken by the cameras. The console in the bottom left corner is used for returning information to the user, and the chat panel in the bottom right enables users to communicate with each other.](image)

The console area provides a single visible place to show all error and status messages. It is used as an alternative to the Java console available in most browsers, as most users either do not know how
to show the Java console, or even if they do, rarely look at it. Robot errors, and any unexpected communication errors are all displayed in the console.

The chat area provides the basic chat interface as discussed earlier in section 8.4.1. A single text field is provided for typing messages; on pressing return, messages are broadcast to all users. Received messages are echoed in the text area below. Hints from the hint server are displayed in the message area along with messages from other users.

8.5.1 The Image Panel

The image panel is the central part of the user interface and operates as both a feedback and input device. The vast majority of the Applet’s functionality is provided via this panel. The panel provides an augmented reality interface to the user which communicates with the robot and camera and workspace modules.

![Image Panel](image_url)

**Figure 8.7: The image panel**

Figure 8.7 shows the image panel running as part of the control Applet under the Windows 98 operating system. Starting from the top, the panel consists of a set of buttons for selecting a camera view, the view itself in the middle, and various controls for changing camera settings.
the bottom. Overlaid on the camera view are various augmented reality elements that assist the user in perceiving the remote environment.

The controls at the bottom of the image map closely to those used in the original CGI interface, in that size, zoom, and quality of images can be changed to suit the user. Additionally the user can specify whether the image will stay fixed about a certain point or will try to move with the robot. Each camera can also be turned off to save bandwidth and speed up response for other cameras.

The middle part shows the latest available image of the workspace, and is updated automatically whenever a new image is received by the camera client. It is also overlaid with various augmented reality elements, which include:

**Stick Cursor** The stick cursor is a tool that allows a user to specify or view up to five degrees of freedom by manipulating different elements. This can be used to generate commands for the robot or to measure objects in the workspace.

**Workspace Point Model** The workspace model consists of sets of points that have been defined in the workspace, these can be points of real objects or reference points to assist orientation or command generation.

**Robot Path** The path the robot gripper has followed can be displayed on the image. This is updated whenever a new robot state is received by the robot client module.

**Axes** The origin and direction of axes can be displayed. A common complaint by users was the difficulty in relating images taken from different angles.

**Measured points and lines** The interface also provides a way for users to measure points and lines in the workspace by specifying points on the image. This can help with establishing how far to move the robot gripper to achieve a task.

**The Stick Cursor**

The stick cursor concept was developed with Harald Friz for use in a stand-alone Applet (Friz, 1998). It was later integrated into the MOM based control Applet as a primary means for controlling the robot. The cursor maps the 5 degrees of freedom of the robot (three for position and two for orientation) to the 2 degrees of freedom of the image. Each line in the cursor represents a degree of freedom. By dragging part of the cursor the user can therefore manipulate each degree of freedom separately. The position of the cursor can be used to form a move command, that can be sent directly to the robot or pasted to the robot command panel. The cursor can also be used to measure position and orientation of objects in the image.

The stick cursor is shown in figure 8.8. It is composed of 6 lines: a *vertical reference line* (VRL), two *horizontal reference lines* (HRL), and two orientation *needles*. The VRL links the *gripper*
Figure 8.8: Elements of the stick cursor. Each element of the cursor can be used to specify a different degree of freedom. Figure a) shows the names of parts of the cursor. The two bottom lines represent the x and y coordinates, the vertical line represents height, and the top two lines spin and tilt. Figure b) shows the directions in which different parts of the cursor can move.

reference point (GRP) with the base point. The base point is the projection of the GRP onto a predefined base plane (the table in this case). The horizontal reference lines are two orthogonal lines through the base point. The endpoints of these lines depict the limits of the robot’s workspace, while their directions depict the X and Y axis. The two needles are used to display spin and tilt and can be interpreted with the metaphor of a needle on a dial plate as shown in figure 8.9. To overcome problems with perception of angles, dial marks are also drawn onto the image while orientation is being changed. An example of the tilt dial being displayed is shown in figure 8.10.

Figure 8.9: Dial plates shown for spin and tilt orientation. These are only superimposed when the operator is changing orientation; the dial marks help with orientation estimation.

The cursor is not just a passive display of the robots pose, it is an interactive tool that enables the user to specify pose by selecting and dragging parts of the cursor. Each element of the cursor is constrained to move in the degree(s) of freedom it represents, shown in figure 8.8 part b). Thus
the GRP which represents height will only move up and down when dragged, while horizontal reference lines will only move in the X and Y directions, and the spin and tilt needles will only rotate in the spin and tilt planes. This scheme allows the operator to directly manipulate the 3D representation of the current pose using only a mouse, without having to change views, or press modifier keys.

**Figure 8.10:** The stick cursor. The operator is changing the tilt orientation. The visible tick marks indicate magnitude and direction of the change. The vertical line to the right of the cursor indicates the current position of the robot.

The cursor is a real object in the workspace from which the 2D image is calculated. Each image’s calibration matrix is used to project the cursor into the image plane and it is then overlaid on the real image from the workspace. This is performed every time the cursor is moved, or the image changes. Similarly, interaction with the stick cursor is also performed on the three dimensional model. Two dimensional cursor positions are mapped into lines in the three dimensional model. Calculations on how the mouse line relates to the stick cursor can then be made by finding how close it is to parts of the stick cursor. The model is then updated and projected back onto the image plane. This happens every time the mouse is moved or clicked. Details of calculations used to implement the stick cursor can be found in Appendix B.
8.5 Client Applet

Other Augmented Reality Cues

A recurring problem that users had with the different camera views was establishing their relative orientation. A greater angle between two views gives better accuracy, but at the cost of increased disparity between the images. To assist users an augmented reality axis was projected onto each camera image. If the real world origin was visible in the image then the axis was placed on top of it, otherwise the cursor was placed in the closest corner of the image to the real world origin. This was most useful for highly zoomed images where the origin and other visual cues such as the table grid, were often out of view.

The robot path was also projected onto the image and updated as soon as new robot poses were received over the robot channel. As the robot pose information was much smaller than the corresponding workspace image, this would arrive first (by a number of seconds for slow links). The path is stored as a series of workspace coordinates that are translated into image coordinates using the image’s calibration matrix which may change with each image update. This gives users feedback on the robot’s state with minimal time delay. It also helps to show them how the robot moves, and how those movements change with different commands. Finally the robot path line provides a point of reference; as the background might move with the gripper, the path shows the new orientation of the camera image.

Workspace measurement

The stick cursor provided a way of measuring the pose of an object, by aligning it manually with an object in each of the camera views and recording its pose. Alternatively the position (but not orientation) of an object can be calculated from its coordinates in a pair of camera views. This was the technique used by the CGI system, where image clicks were recorded, and a robot request formed from finding the closest point between the two epipoles. This was also implemented in the Java interface. Once one image had a point defined, its epipole was drawn in the other images. Once a second point in another image was specified the calculated positions would be output to the Applet’s console. As the camera matrices were known by the Applet this was a very fast local operation. By comparison the CGI version required a separate request and reply from the CGI script for each image click.

The Applet also had a line measurement tool. By specifying two points in the manner described above, the length between two points in the workspace could be calculated. These measuring tools were prompted by previous work in telerobotics (Kim and Stark, 1989; Milgram et al., 1995) and by many users of the CGI system who asked for measurements of the blocks so they could
calculate moves ahead of time. In many teleoperation environments the size of all objects is not known a priori, and they must be inferred in some way. By providing a measurement tool new objects introduced to the workspace can be modelled accurately reducing the need for guess work.

**Point Model**

The natural extension of the two measurement tools described above is to use them to build up a model of the workspace. Spatial information about objects within a workspace is vital for complex robotic tasks, as most tasks require a robot to move, manipulate, or avoid objects. Most available sensors only produce 2D information of a workspace. 3D information of a scene can be recreated by combining data from multiple 2D sensors - for example stereo vision displays (although this information is only fully understandable by a human operator in this form). Machine processing of stereo images combined with range data obtained from other sources can be used to create a range map of a scene, but this still provides a low level of perception. The scene is modelled as a set of interconnected points that can be rotated, translated and displayed in VR or 2D screens but nothing is known about the objects that these points represent. Further perception of the environment can be attempted by recognition of the objects, or at least some properties of the objects; this is known as modelling. Modelling of the environment has some of the following benefits:

- **Programming** - object modelling frees the operator from thinking in Cartesian space by allowing object level commands. Task instructions can thus be at a higher more intuitive language level.

- **View point control** - A model can serve as a graphical aid to the operator by allowing viewing of the robot and its workspace from many more angles than are possible with cameras at the remote scene.

- **Planning** - Storing knowledge of the world in the form of a mathematical model allows automatic planning agents to plan collision free trajectories.

- **Additional information** - Occlusions and other missing data can be filled in using a priori information about a modelled object.

Within a telerobotic framework, there is always a human operator who can assist with the modelling task - this assistance has been referred to as interactive modelling (Milgram et al., 1995) or interactive perception (Backes et al., 1993). Using such techniques, the extent of world knowledge

---

1laser range finders being one notable exception
can be increased by the user gradually developing and refining a quantitative model of portions of the remote world. Even if initially no world model existed, the user can specify the spatial information necessary for defining teleoperation boundaries, obstacles, object handles and endpoints for semi autonomous control.

As a first experiment in modelling a simple point model was implemented. Users could add and delete named points to a global workspace model that was shared by all users. The workspace model was shared amongst users using the workspace model and server discussed earlier in section 8.4.2. Each point in the workspace was projected onto the camera images, using the calibration matrix. Each point was also labelled. These points could then be used in commands to the robot, although the translation from point name to Cartesian coordinates was performed locally by the Applet before sending the command to the robot.

As all users share the same workspace, when one user adds a point to it, it is updated in all the other user’s views. This means that users can collaborate to model the workspace together. Although only one person can control the robot at a time, anybody can access and change the workspace. This open access can have its drawbacks, as a destructive user could delete modelled points that are being used. A compromise solution might require the robot controller to approve changes to the database.

**Block Model**

Block modelling was also investigated, but only tested locally due to time constraints. The same stick cursor concept of section 8.5.1 was extended to allow interactive modelling within a Java Applet. A user could pick from predefined wire-frame models of objects and place them on the workspace image; the models are projected onto the image using the camera matrix. The object’s pose can then be manipulated until the operator judges that it is correctly placed over an image of a real workspace object. This is quite possible in a single monoscopic view of the workspace, although further views assist in reducing placement errors. By adding and placing further objects the operator can gradually build up a model of the workspace.

To manipulate an object it must first be in focus (an object gains focus once the operator has clicked on its centre). Once in focus the familiar stick cursor is shown, position and pose of the object can then be changed using the cursor in the same way that robot pose is changed. An example of interactive modelling is shown in figure 8.11. Three blocks have been correctly modelled and the operator is changing the spin orientation of a fourth.
Figure 8.11: Object identification in progress. The operator has successfully identified a number of objects in the image. The spin angle of the bottom right block is being adjusted - hence the spin dial is shown. The vertical line indicating current robot position is drawn down from the gripper jaws.

A disadvantage of this block modelling technique is that it relies on the size of the blocks being known a priori. Obviously this could be accommodated for by allowing the width, height, and length of the block to be altered as well as its pose. This could be implemented in a similar way to dragging and reshaping of components in drawing packages such as Corel Draw. Alternatively a block can be modelled by specifying four of its corners (see Lloyd et al. (1997)), using the point measuring technique of section 8.5.1. Due to these issues and time constraints the block modelling tool was not incorporated into the Java Applet.

8.5.2 Robot Command Panel

The robot command panel (shown in figure 8.12), provides a graphical interface for the underlying robot module. It has three parts: at the top is a text area for typing and editing commands, in the middle are a set of buttons to control the robot, and at the bottom are a number of fields that show the state of the robot. Commands can be typed directly into the text area, or they can be created from the stick cursor or workspace model in the image panel. This provides a way to fine tune
automatically generated commands. The three control buttons are send, control/release and clear. The send button is used to send any commands in the text area. The control/release button is used to control or release the robot and the clear button clears the text area. The control/release button’s text changes depending on whether the user is in command of the robot. The bottom part of the panel (the status part) has three lines of text. The first line shows the current command being executed, and how many are queued to execute after it has finished. Confident users can work ahead of the robot and queue commands before waiting for the last one to complete. The second line shows the robot state, which consists of its execution state, its pose, and gripper position. This updates every time a new robot pose is received, not just at the beginning or end of a move. The last line shows the nickname of the current controller of the robot, or no controller if the robot is unused.

![Image of robot command panel]

**Figure 8.12:** The robot command panel, with the console below it. The robot is currently executing a command, three commands are queued, and a number have already been completed.
8.5.3 Console Panel

The console panel provides a central point where all parts of the application can report information, updates, and errors to the user. The console is implemented as a Singleton (see Gamma et al., 1994, page 127) that implements a console interface. Messages sent to the console include:

- new controller of robot
- new camera settings
- completion of robot commands
- commands sent by other users
- any errors that cannot be rectified

These messages are designed not only to give users as much feedback as possible, but also to show them what other users are doing. This means that an observer can see the list of commands that the current operator has sent, along with watching how the robot moves in the camera panel.

8.5.4 Using the Applet

A user starts the Applet in the same way they started the CGI system - by following a link to control the robot from the main telerobot web page. They are given a choice between a number of different versions of the Applet, which unfortunately were required due to the many differences between Java Virtual Machines on different browsers and operating systems (see section 8.6.4 for a more in depth discussion of this problem). Once a version has been chosen this downloads a Java version checker (again see section 8.6.4), which in turn loads the Java Applet in the user’s browser.

The Applet itself starts in a separate frame as this allows it to be resized and reduces the chance of the user accidentally stopping it, by pressing the back button in their browser (again see section 8.6.4 for details of this problem). They are then presented with a set of fields that need to be filled in with a username, password and nickname so that they can be logged into the MOM server. The Applet creates a domain root, and attempts to connect to the MOM server, and log the user in. If this is successful then the various client modules within the Applet are configured with the domain root. They then use this to join and subscribe to the various shared contexts they need to be part of. Once this initialisation has finished and the latest robot position and workspace images have been received, then the control interface is displayed in full.

If another user is controlling the robot, then the new user will be notified who the controller is (via the console), and will see commands that the current controller is sending (echoed in the console).
8.6 Results and Experiences

They will receive path updates and new images and any changes in camera settings. They can chat with other connected users via the chat panel. They can drag the cursor around on the images, switch between cameras and switch them on or off. They can also measure and model points in the workspace. However if they are not in control of the robot they are not allowed to send any commands, or change any camera settings (other than turning them off).

To gain control of the robot a request can be submitted using the control robot button. If this is successful they can create and send commands, either directly from the stick cursor or via the robot panel. As commands are executed, both the robot panel, and image panels update asynchronously as new robot states and images are received. This includes updating the status text in the robot panel, drawing the path, and updating images in the image panel.

8.6 Results and Experiences

The MOM system was made publicly available in January 2000. Due to some configuration and hardware problems in February and March it was not available on a 24 hour a day basis until April 2000; it has been running ever since\(^5\). In the period from January to December 2000 over 10,000 user logins were recorded. Section 8.6.1 presents analysis of these user sessions, and compares them with some of the statistics recorded for the CGI system.

A goal of the new framework was enabling collaboration between users. Section 8.6.2 presents evidence that this requirement was satisfied, and discusses how users made use of it. All users of the MOM system were asked to fill out a questionnaire once they had finished using the robot (just over 1000 users filled in the questionnaire). Section 8.6.3 presents some analysis of these answers.

The Applet was shown to work under Windows 95/98/NT, Linux and Mac using Internet Explorer or Netscape Navigator versions 4 and greater. Despite Java’s promise of write once run anywhere, writing an Applet that would work on these different platforms proved very time consuming. Section 8.6.4 discusses these problems in some depth.

\(^5\)The Author left the University (and Australia) at the end of January 2000. All active development of the system stopped at this point. Maintenance and reconfiguration from February onwards was carried out by the author’s supervisor (James Trevelyan) and Sabbia Tilli
8.6.1 Use of the system

In the period from January to December 2000 over 10,000 user logins were recorded for the MOM system. For a user to login to the system, they must have first found the robot home page, followed a link to download the Applet, downloaded it and executed it within a JDK 1.1 or greater Java runtime environment. Finally they must have been able to make a connection back to the server, and then typed in a username to log into the system.

Figure 8.13 shows the number of requests per week at different stages of this sequence of events. The four lines show requests for: the front page, the download page, the Applet download, and the number of logins. As would be expected each consecutive stage has less requests. The biggest fall off is between the front page and the download page - \( \approx 2,000 \) requests per week on the front page translate into only \( 4 \) – \( 500 \) request for the download page, only 25% of front page views go any further\(^6\). The fall off from download page to Applet download is from between the \( 4 \) – \( 500 \) requests down to \( \approx 200 \) – \( 300 \) so about 50%. The number of logins is slightly higher than the number of actual Applet downloads. This may be due to the fact that the jar file is cached locally or on proxy servers, and that users login and use the robot more than once before the cached version is reloaded. The total reduction from the front page to finally logging into the system is about 10%. The large fall off between the front page and the download page may be partly attributable to bad web page design; a number of users commented that they found the robot link hard to find, and the author himself had similar problems after not visiting the site for a number of months. The other reductions could be due to: firewall problems (as shown by the results of chapter 5), user impatience/apathy, or lack of Java support in a browser.

Figure 8.14 shows the distance moved by the robot gripper under the MOM system. This can be compared with figure 4.4 on page 67 which shows the same statistic for the CGI system. The MOM system has a mean of 270 metres per week , which is less that the CGI system. This is most probably due to the reduced number of sessions per week, due to the additional requirements the Applet made on a user’s software and network setup.

Ken Taylor (Taylor, 1999) suggested in his thesis that a method for measuring the usability of a web telerobot was to analyse the distribution of moves in a session. He found this to fit a Weibull distribution for the CGI system. Figure 8.15 shows the results of performing this analysis for the MOM system.

Figure 8.16 shows a histogram of the time between moves made by operators, along with a histogram of the time taken for the actual robot move. The time between moves has a median of

\(^6\)Not all requests will be from interested users, and some may be from search engine robots
25.9 seconds with a lower quartile of 15.8 seconds and an upper quartile of 43.6 seconds. This compares with a median of 44 seconds for the CGI system. The median time between moves has been reduced by 18 seconds (or 40%). The actual execution time or robot moves has a median of 4.5 seconds with a lower quartile of 4.2 seconds and an upper quartile of 5.4 seconds. This means that $\approx 20$ seconds between moves is taken up by network transmission and user thinking time. The time between moves distribution is multi-modal (it has two peaks), whereas the earlier CGI system contained only one. The first peak is at $\approx 5$ seconds which corresponds with the time taken for the robot to execute a move. This peak therefore represents commands that have been queued by a user that execute immediately after each other instead of the normal move and wait sequence. This shows that not only was the system faster to use than the CGI system, but that users also made use of the asynchronous nature of the Applet interface.

The use of the stick cursor was intended to reduce the operator’s reliance on the coordinate system and the visual cues provided by the table grid. The reliance on the visual cues at 100mm spacings along with the strong influence of drop down box options on CGI operators was shown in figure 4.5 on page 68. Figure 8.17 shows the same histogram plotted for the MOM system. Although there are still strong peaks at regular intervals, the number of irregular positions shows an increase over the CGI system. This is especially true for the X, Y, and Z values.
The previous results give a general view of how the robot was being used, and provide a basis for comparison with the CGI version. However to try and introduce a more specific measure of usability and performance of the whole system, more detail is required on what users are doing with the robot. To provide more detail a proximity sensor was placed on the gripper jaws, this would read high if there was something between the gripper jaws. This proximity reading can be
8.6 Results and Experiences

combined with state of the gripper (open or closed), to give a better idea of how the robot is being used. The four states can be named as follows:

- **open proximity**— A free standing block is between the jaws. Either the operator has just stacked a block, or has just moved the gripper over one.
- **closed proximity**— The gripper is closed with an object between the jaws.

**Figure 8.16:** Histograms of the time between moves, and robot command execution time. The time between moves has a median of 25.9 seconds with a lower quartile of 15.8 seconds and an upper quartile of 43.6 seconds.
Figure 8.17: Histogram of robot poses moved to by operators using the MOM based Applet interface. The X, Y and Z values show more variation indicating a reduction in the dependence on visual cues and interface restrictions.

- **open clear** – The gripper is open, and nothing is between the jaws. The user is possibly searching for an object to pick up.
- **closed clear** – The gripper is closed, and nothing is between the jaws.

From the perspective of the block stacking task. The most interesting cases are the **open proximity** and **closed proximity** states. The **open proximity** state implies that a there is a free standing object
at the height of the gripper. Obviously if a user is stacking a series of blocks then this state will be obtained more often. Users that are not able to stack blocks will not reach this state by accident, and the moves will tend to produce lots of log entries with the proximity sensor reading clear.

<table>
<thead>
<tr>
<th>gripper</th>
<th>proximity</th>
<th>height (mm)</th>
<th>CGI (5,000 moves)</th>
<th>MOM (first 25,000)</th>
<th>MOM (last 25,000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>yes</td>
<td>Any</td>
<td>11%</td>
<td>10%</td>
<td>30%</td>
</tr>
<tr>
<td>open</td>
<td>yes</td>
<td>&gt; 50</td>
<td>1.8%</td>
<td>1.5%</td>
<td>11%</td>
</tr>
<tr>
<td>open</td>
<td>yes</td>
<td>&gt;100</td>
<td>0.7%</td>
<td>0.6%</td>
<td>5%</td>
</tr>
<tr>
<td>closed</td>
<td>yes</td>
<td>Any</td>
<td>28%</td>
<td>25%</td>
<td>30%</td>
</tr>
<tr>
<td>open</td>
<td>no</td>
<td>Any</td>
<td>47%</td>
<td>45%</td>
<td>24%</td>
</tr>
<tr>
<td>closed</td>
<td>no</td>
<td>Any</td>
<td>15%</td>
<td>20%</td>
<td>15%</td>
</tr>
</tbody>
</table>

Table 8.1: Types of gripper states at the end of a robot move. Constructive moves are those for which the proximity sensor reads a positive signal. The percentage of moves in which the gripper was open, and the proximity sensor set increased from ≈ 10% with the CGI system to 30% with MOM system

Table 8.1 shows results of analysing moves for the CGI system, the MOM system - when it was first made public - and the MOM system again in December 2000. The first row shows the percentage of moves which ended in the open proximity state. Row two adds the requirement that the height of the gripper is greater than 50, and row three requires a height of more than 100. In all three cases the final MOM system figures show a large increase over the CGI and early MOM figures, suggesting that users were indeed stacking more blocks once they got used to the new system. As would be expected in all cases the percentage of moves decreases as the height threshold is increased. The remaining rows show the percentage of times the other three gripper states were obtained.

A final test of whether the MOM system was preferred over the Java system was to ask users; this was done via a questionnaire which is discussed in detail in section 8.6.3. One of the questions was whether users preferred the Java or CGI versions - 43% said they preferred the Java systems, 4% preferred the CGI system. The majority (54%) did not answer, this was probably due to the fact that the majority of users are first time users and would never have seen the CGI system operating. However for those that did answer, the Java interface seems to have been the preferred interface.

8.6.2 User Collaboration

Users could collaborate, share and learn how to use the robot in a number of different ways in the Java interface. This section analyses some of the log files to try and establish whether this aspect of the system worked, and if users found it useful.

---

7 Due to sensor problems only a small amount of CGI data is available. The proximity sensor was introduced in August 1999, but the author was working with ABB robotics in Norway at the time and was unable to fix the problem until late 1999. The CGI data presented in this table is for a month before the MOM system was introduced.
New controller of the robot: bryan H
bryan H sent
bryan H sent
New settings for camera YCam
bryan H sent rz10
bryan H sent x=305 y=460 z=100
bryan H sent x461, y195, s45, t40, z100
bryan H sent gclosed
bryan H sent gopen
bryan H sent z=20
bryan H sent z=5
New settings for camera YCam
bryan H sent z=100
The robot is free

Figure 8.18: A short investigative session by a user, seen from the console window of another user. All commands (robot and image) that a robot operator sends are displayed on each user’s console.

To learn to use the robot they had a number of options:

- Watch other users
- Read the hints
- Work it out themselves
- Ask other users
- Read the (out of date) documentation on the website.

The first two options required no intervention by the user, as the hints were echoed in the chat window, and any robot commands sent by other users were echoed in the console window. Figure 8.18 shows an example of the type of information echoed to an observer’s console window. This can be viewed in conjunction with the image and robot pose information to gain an idea of how commands are interpreted by the robot. As the robot moves the status and images would be updated on the observer’s screen as they would be with the current operator’s. Without being in control of the robot, users could also experiment with dragging the stick cursor. To take this a step further a user would need to gain control of the robot, and start sending real commands and use the image and state feedback information to learn how the robot operated.

The chat system provided an interface for users to ask other users for help, negotiate for control of the robot, complain, or even discuss totally unrelated topics. The chat console also provided help in the form of hint messages. Appendix C section C.1 contains a number of selected chat conversations, these have had the intervening automatic hints removed but are otherwise reproduced as they occurred (typing and spelling mistakes included). Email addresses typed by users have been removed to protect their privacy. Some of the types of conversations include:

- General discussions (C.1).
- Experienced users helping new users (C.3, C.6).
- Accusation of other users (C.2).
8.6 Results and Experiences

- Negotiation for control (C.5).
- Talking with the hints robot (C.7).
- Abusive comments (not reproduced here).
- Conversations in other languages - French, Spanish, German.

These conversations show that the chat system was used by users to communicate with each other. This took many forms, from collaboration in the form of constructive help, to negotiation, to abuse. It was amazing how informative and helpful some users were, especially considering that there was limited help available on the system. The chat system also seems to have been used to negotiate control of the robot; an unwritten rule evolved that it was bad manners to take control of the robot without asking first (see conversation C.4). As surprising as how polite and helpful users could be it was also disappointing how abusive some others (see conversation C.5 for a tame example) could be. This demonstrates one of the problems of having a system that is freely available on the web: there is no guarantee that people will use the system in the way that you intended. Quite often new users seemed to think that the chat server was a real person; this led to some quite amusing conversations such as conversation C.7.

The workspace modelling aspect of the interface was not heavily used. This was probably due to the fact that only points could be modelled. Modelling points did not really speed up execution of commands, or provide a higher level of control, so there was little motivation for users to use it. Those that did tended to use them for labelling areas of the workspace for easy reference, between moves, and between views. Some users used the fact that points included a label as a form of Augmented Graffiti, either to grab the attention of other users, or to leave evidence of their visit. Extending the modelling and robot language to include objects instead of points would be a very interesting area for further work.

8.6.3 User Feedback

To assess the user experience of the telerobot system users were asked to fill in a short questionnaire when they had finished using the MOM system. This consisted of the following questions:

1. What operating system are you using?
2. What connection do you have to the Internet?
3. What CPU speed is your computer?
4. How much delay was there using the robot?
5. Do you prefer the Java version or the CGI version?
6. Did you collaborate/chat with other users?
7. Do you have a technical background?
8. Have you ever seen a robot?
9. Have you ever controlled a robot (other than a telerobot)?
10. Do you know what a Quaternion is?

Figure 8.19 shows the results of these questions, generated from 1152 returned questionnaires. Chart a) shows a breakdown of answers to the connection type question, showing that most users have at least a 56Kb/s modem and almost 50% have something faster; this is encouraging for future applications as faster connections will reduce the communication delay in both directions. Plot b) also shows that almost all users have a 200MHz processor, with half having at least a 400MHz processor; this is again encouraging as Java based interfaces will require more powerful processors to overcome the speed reduction due to it being an interpreted language. Plot d) shows the delay that users estimated they experienced; although a lot of users didn’t answer this question 50% experienced a delay of under 5 seconds, only 6% said that the delay made the system unusable. Chart e) shows whether users collaborated or chatted with users; only 19% said that they did, but although this number is small it is still 19% more than were able to collaborate with the old CGI system, and it does show that people used this feature of the system despite the fact that the interface only implemented the most basic collaborative functions that the framework made possible. Finally chart f) shows the answers to the final four questions. These were designed to get an idea of the technical background of the telerobot users. Over 50% of users had a technical background and over 70% claimed to have seen a robot, and 40% claimed to have controlled a robot. These figures seem remarkably high. Only 15% knew what a Quaternion was, suggesting that very few users had in depth knowledge of robot technology.

Of those users that filled in the questionnaire over half left comments on the robot. Section C.2 shows a selection of some of these comments. These comments cover a broad range of subjects from praise (C.15, C.30), to complaints (C.2, C.5), to suggestions (C.9, C.35). The comments which include constructive complaints and suggestions are most useful and indicate some of the problems of getting the system to work effectively. For example there were quite a few complaints about the speed of the system (C.13, C.21). What the comments also show is that it is probably not possible to have one interface that is good for everyone. Some people liked the cursor interface C.17, while others found it hard to use C.29. Similarly some found the hints useful C.8 while others didn’t like them (conversation C.7). However this level of user feedback shows how useful having an interface available on the web can be for establishing which aspects of the system users like and dislike.
8.6 Results and Experiences

Figure 8.19: Questionnaire results
8.6.4 Applet practicalities

A significant and unexpected amount of time was spent trying to make the control Applet work under different network setups, operating systems, and browsers. Although the idea of running Java Applets in a client browser is very attractive, there are a number of hurdles to be overcome:

- Security restrictions
- Distribution
- Runtime limitations
- Different Java runtime environments
- Remote debugging

Applets must obey a number of security restrictions known as the Java sandbox. For an Applet to be able to operate outside the sandbox it must ask for permission from the user. In order to ask permission, the Applet must be cryptographically signed. Although this is in principle an excellent scheme that offers far more protection to users than other technologies such as ActiveX, differences in implementation across vendors mean that the solution is much more complex. For example although the default sandbox security settings should allow an Applet to receive UDP packets, this is not possible in Internet Explorer (as explained in section 5.3.2 on page 87). The default sandbox also leads to problems with RMI callbacks (as explained in section 7.4.2 on page 139), despite the fact that RMI is Java’s standard remote procedure call interface. Both these problems can be addressed by signing an Applet, but this also has its problems. Java 1.1 did not define a single standard for signing an Applet and therefore two separate, inconsistent systems have evolved, for each of the major browsers. The Java 2 platform introduced a signing standard, but the default Java Virtual Machines of version 4 browsers only support Java 1.1. Due to these signing complications it is preferable to avoid API calls that require extra security, so that the Applet can operate untrusted. This can mean some sub-optimal work arounds in the UDP and RMI cases mentioned above.

Applets are distributed by embedding Applet tags in a web page, which reference a location for the Applet code to be downloaded from. A Java Applet will not start running until this code has been downloaded and verified. A Java Applet is made up of a number of class files, that can be downloaded separately - however this becomes very slow for complex Applets which consist of hundreds or even thousands of classes. To avoid this problem classes for a specific Applet are usually packaged into JAR files so that all files can be downloaded in a single connection. There are actually 3 extensions for Java archives, these are .zip, .jar and .cab. The .zip and .jar formats are exactly the same internally and differ only in their extension; .cab files are a Microsoft format and are only supported by Internet Explorer. It should be noted that making an Applet into a JAR file is not the same as C/C++’s linking mechanism where all required symbols are checked and
linked into a single executable. A Java archive is merely a collection of class files; it is up to the programmer to provide the correct files. The Sun Java development kit provides no easy way to do this. There is a Jar tool for packaging files, but which files to include must be specified on the command line. The easiest solution is therefore to include all known class files to ensure that all required files are included. This can result in Jar files that are significantly larger than they need to be. There are a number of free tools available that make the process of packaging Jar files simpler and more effective.

The first is a Java program that recursively inspects Java files for dependencies and collects all of them into a single Jar file, thus ensuring that only those classes referenced are included. However classes referenced dynamically in code using the `Class.forName()` function call are not included, and must be included on the command line. Thus creation of a complete archive requires the main class and and dynamically invoked classes to be included on the command line. For a large project this is generally a case of trial and error, which involves packaging the file, and then running it looking for exceptions caused by classes not being found. If this procedure is followed then there is still no guarantee that all class invocations have been found, as names of classes can be created at runtime. This is both a strength and weakness of Java: its dynamic class loading offers great run time flexibility, but at the same time introduces fragility and makes the concept of runtime linking impossible.

An Applet's JAR file may contain code that is never used, which can be omitted. There may also be optimisations that can be made on the basis that the class will only operate with other classes in the same JAR. This can reduce the size of the JAR as well as improve runtime performance of the Applet. One such tool that can perform some of these optimisations is the IBM Java tool JAX. By using JAX on the interface code it was possible to reduce the JAR size by half to only 130KB, less than a minute download for a 56Kb/s modem. JAX performs the same task as the packager above, but also performs some optimisations and obfuscation. The optimisations improve run time speed, by techniques such as finalising classes that are not extended, and reducing size by removing unreached methods and unaccessed fields. This optimisation comes at a price: often the resulting classes are incompatible with their uncompressed originals - of particular importance when objects are being exchanged with other Java processes. Optimisation can be controlled on a per class basis, so that classes are compatible. But as in the above case this must be performed using trial and error, and exhaustive testing is the only way to ensure that all classes are compatible and the the application will work as intended.

The Applet was made available in both the JAR and JAX versions. The JAX version was about half the size of the normal JAR file, and seemed to offer similar stability. Exactly how stable the respective versions are is hard to access as there are so many other inconsistencies in users’ Java
setups. For example the JAX version crashes Netscape 4.08 under Windows 98, but runs fine with the same browser on Windows NT. To allow users with non JDK 1.1 compliant browsers to use the Applet it was also provided in a Java Plugin form. To provide the Applet for use with the Java plugin, no changes are made to the Java code. All that is changed is the HTML code to launch it. As a result, to provide versions to work on as many platform and network configurations as possible, four very similar HTML pages need to be maintained.

There are a number of constraints on an Applets runtime behaviour. An Applet may be run on many different hardware configurations, which may have slow processors or limited memory resources. Java applications tend to use more memory than their C++ counterparts. For example the image server used 4MB of memory as a C++ application but once a Java front end was added using JNI, this increased to 16MB. Applets must be designed with memory and processor requirements kept to a minimum. Other constraints include the different resolutions that users have; the Applet was designed for an 800x600 screen but this meant that useful space was wasted for larger dimensions. This resulted in problems with having to scroll larger images (see comment C.6), and meant the fonts were too small on some systems (see comment C.36). This again shows that it is very hard to produce something that works for all setups, and that multiple versions may be required depending on the target audience.

Applets must also exist within the page oriented environment of a browser. This means that if a user moves from the page that the Applet was loaded from then the Applet will be destroyed. All Applets must implement init, start, and stop methods which will get called as the result of different activities in the browser window. This can cause problems for applications such as the telerobot where session should be persistent and continuously connected. Additionally, different browsers behave in different ways, calling the Applets start, stop methods at different times, meaning that more additional redundant code is required to cope with all eventualities.

All Java Applets run within a Java Virtual Machine; different browsers have their own VMs. Although Java promises write once run anywhere code, in reality the burden of producing cross platform code has shifted from the application writer to the virtual machine writer. These virtual machines often do not behave in the same way. For example Internet Explorer closes socket connections after some time, whereas Netscape keeps them open. A popup menu that works fine in Internet Explorer, and some versions of Netscape fails to work in Netscape 4.08 (see comment C.24). This problem will hopefully decrease over time, as Java becomes a more mature technology, but highlights the fact that the cross platform problem is as yet far from solved.

Some Java implementations also have stability problems. The Virtual Machines (VMs) shipped with the Netscape and Internet Explorer browsers, although fine for lightweight, periodic use,
prove less stable when operated for longer periods, often resulting in the browser, or even the complete machine locking up. A possible solution to this problem is the use of the Java plugin, although this requires a download of the Java runtime environment. This is over 12MB which is a significant time commitment for users with slow modems. Web users are notoriously fickle in their attention span, so any extra requirements such as this are likely to reduce the use of the system.

### 8.7 Calibration Application

In the CGI version of the system, calibration of the cameras was a laborious and error process that took an hour to complete. The robot had to be jogged to set positions, a full image taken, and the process repeated for about 20 points. The location of the gripper was then recorded manually for each image, using an image editing package such as Photoshop or Paint Shop Pro. Finally a calibration program was run on the points and a calibration matrix produced, that could then be used to relate image coordinates to robot coordinates. Errors were frequently made in either jogging the robot, or identifying the location of the gripper in the image. As a result calibration was not performed as often or as accurately as it should have been.

To test the flexibility of the MOM framework and to automate this procedure a Java calibration program was implemented. The application joins the MOM server as another peer in the distributed system. This means that the calibration can be performed while the whole system is running, without the need to shutdown, or restart any of the servers.

The calibration application subscribes to the image channel it is calibrating and to the robot token and channel. It logs in with a high authority login, so that control of the the robot token is ensured. Once logged in, a set of calibration points is read in from a file, the robot token is grabbed, and the camera settings are set to record a full frame at high quality. Robot commands are then sent via the robot channel to move the robot to each of the points in turn. At the end of each move, a full frame image is taken by the image server, and returned to the calibration user over the image channel. For each move the user must then click the endpoint of the gripper in the image to record its $u, v$ coordinates for the calibration algorithm (Jain et al., 1995). Once all the points have been covered, the calibration is performed using the list of image points, and robot coordinates. The calculated calibration file is then displayed. An error estimate is given for each point, and if there look to be any large discrepancies, the robot can easily be moved back to the position in question. New images can be taken to check the recorded point. Once the user is happy with the calibration data the new calibration data is saved to disk. Closing the application logs out the calibrator and
releases control of the robot token. Figure 8.20 shows the calibration application running under Windows 98.

![Image Displayr](image)

Figure 8.20: The (Java) calibration application

The image sever could be configured to notice new calibration files, but to safeguard against accidental recalibration it must be restarted to read new calibration files. This is a simple process, and as all other peers are connected to the MOM, is completely transparent to them, as the only connection that is affected is the one between the image server and the MOM. The whole calibration process takes about 10 minutes, and can be performed while other users are connected to the system - they can be notified as to what is going on via the chat interface. For even faster calibration the robot speed restriction imposed on normal operators could be lifted for trusted calibration users. This would require a modification to the robot server to include a profile of registered users, and their level of expertise, which would relate to a maximum allowed robot speed (see section 9.4).
As the calibration application runs as just another framework peer there is no need for the application to be run locally in the robot lab. In fact figure 8.20 was taken with the application running on the author’s laptop in London, while connected and calibrating the robot back in Perth. This perhaps is a more immediate commercial example of the applications of web based control.

8.8 UWA Telelabs

The previous examples have show how the MOM framework was applied to the UWA telerobot system. This section considers how the framework might be applied to the UWA Telelabs project (Trevelyan, 2000). The Telelabs project is an initiative at UWA to make some undergraduate engineering laboratory apparatus available on the web. This will not replace normal practical sessions, but will allow students to complete lab work in their own time as the labs will be available beyond normal working hours. Some of the labs planned include:

**Electric Iron** Students perform experiments to model the operation of an electric iron. The student must be able to control the iron, and a fan that blows onto the face of the iron. Feedback is via thermocouples on the base of the iron, and a camera that shows whether the fan is on (via streamers).

**Torsional Vibration** Students drive a two degree of freedom torsional vibration rig and observe the excitation of various parts of the rig.

**Sandbagger** Students program an automatic sand bagging device using state diagrams and discrete control techniques. The student must be able to program, run and observe the device in operation.

In each of these experiments, students will need to perform some experiments, by controlling different parts of the apparatus, and they will then need to record and analyse some of the results and submit a lab write up. The actual control and observation of the apparatus may be done on a computer in the same room, from a lecture room by a demonstrator, or even from the student’s home. Some parts of the lab experiment will be performed concurrently by a group of students all of whom may be logged in via different computers. Access to the lab, and recording of students activities will need to be controlled via user accounts and groups that will define when, who, and how the labs can be used.
8.8.1 Applying the framework

Each lab would have its own domain that might be part of top level Telelabs domain. Each lab domain would have a number of channels and tokens that would be used to represent the various devices in the experiment. For example, the iron experiment might have a channel for each thermocouple, a channel for the camera images, a channel for the fan control, and a channel for the iron control. A single token could be used to define who is currently allowed to change the fan and iron settings, or there might be separate tokens so that each could be controlled independently. Each domain would probably have its own chat channel as well so that students can exchange ideas.

An important requirement of the Telelabs is managing access control. Different users will have different rights within the system, and each lab may have its own requirements. As each shared context within the framework can have a manager, each lab can create its own contexts and managers that implement the schemes that it needs. In some cases not all management decisions can be made algorithmically and a human moderator may be needed to make certain decisions. This would require a special manager object that would create its own moderation channel which it would use to delegate requests. The moderation channel would in turn have its own manager which would only let users with moderation status join it. This would allow multiple moderators if required.

For example, one lab requirement is that at a given time, only students that are scheduled for a lab should be allowed to join it. However, if a student who is not from the lab requests to join the lab, the current students should be asked if they will let the student join. Within the lab, any student should be able to control the devices. To satisfy this requirement a lab domain, moderated manager, moderation channel, control token, and channels for each device would be required. The lab domain would be created with a manager that has access to the lab timetable. This manager would create a moderation channel within the domain that anyone can join. As the moderation channel is within the lab domain, to join it a user will have to have joined the lab domain first. To join the lab a student would login to the MOM, and then try and join the lab domain, the domain manager would check the student’s id against the lab timetable. If the student is in the current class then they will be allowed to join immediately. If they are not, the lab manager will send out a query on the moderation channel; if a moderator replies saying that the student may join then they will be allowed to join, otherwise their request to join will be refused. Once students have joined the lab domain, they can then join the moderation channel, the token, and any other channels. As the requirement for control of devices within the lab is that it be free access, the lab token will have a manager that allows anyone to grab it (not forgetting that the parent domain has already taken
care of making sure that only valid students are using the lab). In this case the token is being used merely as a form of mutex to ensure that only one user at a time is trying to control the devices.

Another requirement is that users might have different control status once joined to a lab:

- Observe only - user can watch, but not control anything
- Control if free - user can control provided the device is free
- Exclusive control - user has exclusive control over device
- Private control - as above, but no other users can observe

The first three requirements can be fulfilled using token control similar to that used for the telerobot. Users have an access level, and whenever they try to grab a control token, their access level is compared to the current owner. Those with observer only status would have an access level of 0 meaning that their request would never be granted; the control if free users would have a level of 1; and the exclusive control users would have a level of 2. The fourth requirement (private control) would need to be implemented by the domain manager. Once a user with private control joins the lab, all other users would have to be removed from the domain, and future join requests rejected.

The feedback and control of labs may approach closed loop control instead of the supervisory control used by the telerobot. In this case all data sent within the system needs to be time-stamped so that delays can be calculated exactly. As all parcels sent within the MOM are time-stamped this requirement should be met with no new code required.

Users of the system are likely to be on widely differing networks, from local users such as students in labs or lecturers, to students at home using a modem, to overseas users. LAN users may take advantage of the lack of firewalls to use UDP for high bandwidth data, whereas remote users would have to use TCP, and may even have to tunnel over HTTP if behind a very restrictive firewall. The MOM’s ability to use multiple different types of transceivers concurrently would mean that all these users could be connected at the same time, without being aware that they are using different protocols to communicate.

Hopefully these examples show that with different combinations of domains, managers, channels, tokens and transceivers that many varied collaborative control applications can be developed. New managers will need to be designed to implement these schemes, but the core part of the framework - the domains, tokens, and channels - require no modification, as does the MOM server itself. Obviously the student interface code, and device control code would also need to be written, but the framework would provide the collaborative glue to join students and devices together hopefully making these parts more reliable and much easier to write.
CHAPTER 9

Discussion

9.1 Summary

This thesis documents two different telerobot architectures that were designed and built to be operated over the web. They were used over a period of four years by thousands of users. The first system used CGI to provide a static HTML interface with images of the robot and controls for sending commands. The second system used a Java Applet which was connected to the rest of the system via a distributed framework, which not only allowed control of a robot, but also allowed users to communicate and collaborate with each other.

The CGI system was an evolution of Taylor’s work (Taylor, 1999). On visiting the website and choosing to control the robot, users were presented with an HTML page showing images of the robot and a set of controls to send commands to the robot. As this interface was an HTML page it required minimal hardware to run. Any computer with a web browser and Internet connection could control the robot. The system was popular with nearly a thousand users a week operating the system. Some amazing structures were built despite the limited nature of the interface.

There were a number of other problems with this system. It was slow to use, as each change on the client interface required a new HTML page to be generated. These new HTML pages were generated by the CGI executable at the remote site, which had to be run every time a request was received. On the client side the interface was constrained by the limitations of HTML and the page based interaction it imposed. Communication was also restricted to the HTTP protocol which provides no way of initiating communication with the client.

On the server side there were problems of session tracking, and state management due to the transient nature of both HTTP connections, and the CGI executable itself. State management was particularly difficult as the state was distributed in different places. State is cached by both the
page showing in a user’s browser and by the CGI script. There is no way of keeping these states
synchronised between client requests.

Some of these issues can be alleviated with the use of client side extensions to standard HTML;
such as Javascript, frames and dynamic HTML, however the interface is still fundamentally con-
strained to operate as a set of static pages. In order to avoid these problems a new type of interface
was required.

By using Java for the client interface many of the above issues can be addressed. Java provides
complete control over how an application looks; it can perform local processing avoiding unnec-
essary contact with the remote site; and it allows the use of different network protocols1. For this
reason the second architecture built used a Java based interface.

Before deciding what protocol to use to communicate with the Java Applet a number of network
tests were performed. The purpose of these tests was to establish the network characteristics of
typical users. These tests were performed by an Applet distributed with a web page. The tests
included sending different sizes of packets to establish throughput and latency of connections, as
well as trying both UDP and TCP connections. It was found that very few users were able to
communicate using UDP, due to firewall and Java implementation problems. Round trip latency
was found to be less than a second for most connections. Bandwidth varied immensely with the
minimum bandwidth being around 1KByte per second.

Various network protocols and architectures were explored, including the use of RPC and its close
cousin RMI. Collaborative toolkits such as the JSDT and IBM’s shared data objects were also
considered. However no software was found that suited the requirements of an Applet based col-
laborative control system. The requirements for such a framework include minimal size, no license
fees, and the ability to use multiple network protocols to adapt to different network environments.
The framework should also provide user authentication, session tracking, asynchronous (event no-
tification) and synchronous communication (remote procedure execution), resource control, and
prioritisation between control and data streams. Considering how low the bandwidth was for some
connections it is very important to keep low bandwidth control streams (commands, replies, and
events), in separate streams to high bandwidth data such as images.

A framework based on Message Oriented Middleware (MOM) was developed to provide the basis
for the Java based architecture. This consisted of a central MOM server that clients (known as
peers) maintained a permanent connection with. The MOM server maintained a list of shared
contexts that connected peers could create, join, and leave. These contexts consisted of three types;

1within certain security restrictions
domains, channels and tokens. Domains contain and manage other contexts. Channels represent different communication routes and can be used for the exchange of messages. Tokens are used for resource control. All these contexts can be managed independently to control how different parts of the distributed application interact.

The MOM architecture was applied to the UWA Telerobot, and was been run for a year and used by thousands of users. The system halved the time between moves made by users, to a median of 20 seconds, and also showed an increase in the number of constructive moves that users made. It also showed some of the collaboration facilities of the framework in operation, the most well used of these being a chat interface.

9.2 Contributions

The list of contributions this thesis makes to the field of web Telerobotics can be summarised as follows:

- The identification and discussion of techniques available for web telerobotics. This includes techniques and protocols such as CGI, HTTP, HTML, Java, RPC, and MOM.
- Comprehensive testing and analysis of network characteristics of web users around the world. This includes an estimate of delay, bandwidth and network restrictions due to the use of firewalls.
- The identification of requirements for a web control framework. These include: user authentication and session tracking, synchronous and asynchronous communication, resource allocation and management, and support for multiple communication protocols. All of which must be provided within the security and network constraints imposed on Java Applets.
- Implementation of these requirements in the form of a reusable cross platform framework that will work on any operating system that supports Java. The framework has been used for a year to support a Java based telerobot interface.

This thesis also includes contributions which are specific to the hardware, and task chosen. These contributions include:

- A multiuser collaborative control Applet that enables web users to control an ABB 1400 robot. The Applet makes use of the framework to provide event notification, messaging, remote execution, and resource control. The Applet makes use of Augmented Reality to
edit and display a shared workspace. An Augmented Reality cursor is used to act as an input device for forming robot commands.

- A robot server application that will work with any ABB robot controlled by an S4 controller as demonstrated by work in Norway with an ABB 2400 robot.
- A calibration application that can be operated over the web so that a robot and camera may be calibrated remotely.
- An alternative much simpler CGI system that can be used to control a robot over the web.

9.3 Conclusions

The results of the MOM system show that users not only preferred it, but were able to make moves faster, and to be more productive. They were also able to collaborate with other users via the chat and workspace modelling tool. New users could learn passively by watching the commands that users sent, observing the robot move, asking questions via the chat interface and reading the hints messages.

An unanticipated result of using the framework was that a robot and camera calibration application was relatively easy to implement. The application used the framework to move the robot and take images. The application could login to the system and calibrate a camera by moving the robot to a number of set positions. This could be done while users were logged in. As this application was just another MOM peer it could be run remotely, as indeed it was by the author from the UK.

The use of the framework for the last year without any major problems validates its potential as a basis for a web system. Its provision of services such as user authentication, resource management, asynchronous and synchronous communication, and its ability to use multiple network protocols mean that it could be used in a diverse range of web control applications.

Although the final MOM system was preferred over the original system, the number of people that used the Java system was lower than for the CGI system. This can be explained by the extra Java download step required, and the need to connect to a socket on an unprivileged port. Any extra stages required to run a web application will lead to a reduction the number of users.

Both architectures do have their applications. The CGI system is more suited to simple applications that can easily be mapped to a page based view such as the Xavier robot which has a high
level of autonomy. For other systems the MOM framework offers more services and greater flexibility. It also makes web controlled devices easier to implement, as the core services required are already provided. The framework is particularly suited to collaboration between geographically distributed systems, some of which may be other autonomous agents, or users.

The development of the Applet was not without its problems, due to variations in implementations of Java Virtual machines on different operating systems and browsers. A process of experimentation and user feedback was required to find solutions to problems on different platforms. These problems were far less of an issue with the simpler CGI interface and are a result of interface complexity. A more complex interface will tend to be less portable as it requires more functionality from the virtual machine/browser/operating system that it runs on. A simple interface like the CGI interface will work on just about any browser, it was even possible to control the robot from lynx, a text based browser and from Web TV.

Complexity requires more client processing power too. The CGI interface has minimal processor and memory requirements, while the Java Applet interface requires at least a 200MHz processor speed and 32MBytes of memory. These requirements are quite modest by today’s standards but they were kept intentionally low and certain features such as Java 3D were deliberately omitted from the interface. The CGI interface is also fairly instant as it is loaded in the time taken to download a web page, whereas the Java Applet must be downloaded before it can be used; this can take a number of minutes\(^2\)

\(^2\)With the use of caches and protocols such as the Java Network Launch Protocol (JNLP), this can be reduced to a first time only requirement.

\(^3\)known as HTTP tunnelling

9.4 Future Work

There are a number of exciting possibilities for future work, some of these relate to the core MOM framework, others are related to the UWA Telerobot system, and a third category is new web applications that make use of the MOM framework.

The main area for future development of the framework is to extend the current transceivers to cope with more varied network conditions. This includes handling a loss of network connection, so that a user can re-attach to the MOM without other parts of the system realising that there was a break in the connection. There also needs to be a transceiver that will work over HTTP\(^3\) for those users that are behind a firewall. A shortcoming of the current implementation is the fact that tokens cannot be reserved/granted for a period of time, this would be necessary to enable scheduling of
resources for different student classes for example. Optimisation and stress testing of the MOM server is also required before it can be used in more demanding environments.

The telerobot application can be extended in many ways. The modelling tool is currently very crude and only allows a point model to be shared amongst users. This could be extended to model blocks and other objects which could then be the subject of commands. This would provide a more interesting study of how users might cooperate. A more complete model would also provide opportunities to use other visualisation techniques such as virtual reality to provide the users with alternative views of the workspace.

Of course modelling of the workspace does not have to be performed by users; there is nothing to stop an application or agent logging into the system to perform machine vision on some of the images. This application might be developed and run by a different research group, or there might even be a group of such agents from many different research groups.

Although creating an Agent to replace the human operator completely is not feasible today, it is possible to provide helper applications that can perform part of the cognitive process. These helpers might provide some scene measurement, using calibration information, or they may be able to locate certain types of objects. A most useful helper would be that of visual servo control, where the Agent takes control of the robot to align two objects in the workspace. Once the servo helper has finished aligning the objects it would return control back to the original user. Although this would be slower than a purpose built image servoing system, it would probably be faster than the current move and wait strategy adopted by users.

Although Java does free the interface from the confines of a browser window, it is still controlled by the browser and this can cause some problems. There is now a standard called the Java Network Launch Protocol (JNLP), which still uses the web for distribution of applications, but executes them within their own environment. Applications that use this protocol can still have the benefit of being distributed via the web, but at the same time can at last be truly free of the web browser.

Different forms of telerobotic control could also be implemented. In practice many telerobotic tasks cannot be performed with position control alone. Force control and compliance are required for tasks such as insertion and assembly. Any new control scheme would need to work within the constraints of supervisory control with variable time delay.

A first step might be to introduce force sensing and compliance without any active force control. Additional parameters to commands would be used to add constraints to moves. For example to place one block on to another an operator would send a move command with a constraint that the
vertical force should not exceed some threshold. The move would stop once the threshold had been reached (when the two blocks are touching). Certain combinations of constraints would be repeatedly required and could be packaged as named commands that the operator could select.

Future work might also concentrate on applying the system to other applications. Education provides one of the more immediate applications for web telerobotics and control. A number of schools in America and Europe were known to use the telerobot for teaching, some even had a special user name to guarantee them control. UWA itself is currently investigating the use of web control to allow students to perform some student labs online. Application of the framework to provide the basis of these labs would be an excellent test, and would establish it has really satisfied its goals of providing the set of services required for web control of devices.
Bibliography


Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ActiveX</strong></td>
<td>An object-oriented technology from Microsoft that relies on the Component Object Model (COM). The most common use of ActiveX is for the creation of reusable code that can be used in other applications.</td>
</tr>
<tr>
<td><strong>API</strong></td>
<td>Application Programmer Interface. The interface a programmer would use to use/plugin to another library or application.</td>
</tr>
<tr>
<td><strong>ARQ</strong></td>
<td>Automatic Repeat Request. A communications feature where the receiver asks the transmitter to resend a block or frame because errors were detected by the receiver.</td>
</tr>
<tr>
<td><strong>ASCII</strong></td>
<td>American Standard Code for Information Interchange. ASCII is a code for representing English characters as numbers, with each letter assigned a number from 0 to 127.</td>
</tr>
<tr>
<td><strong>ATM</strong></td>
<td>Asynchronous Transfer Mode. A networking technology that contains a flexible multiplexing and switching technique which provides variable bandwidth for local-area and wide-area networks. Unlike ordinary synchronous configurations, ATM permits flexible allocation of available bandwidth for data, voice, images and video.</td>
</tr>
<tr>
<td><strong>CGI</strong></td>
<td>Common Gateway Interface. The technique used to launch processes to generate dynamic content from a webserver. See <a href="http://www.w3.org/CGI/">http://www.w3.org/CGI/</a>.</td>
</tr>
<tr>
<td><strong>CORBA</strong></td>
<td>Common Object Request Broker Architecture. CORBA is an industry wide standard for building distributed object oriented applications that are cross platform, and cross language. The standard is managed by the OMG.</td>
</tr>
<tr>
<td><strong>CSIRO</strong></td>
<td>Commonwealth Scientific and Industrial Research Organisation. An Australian scientific research institution</td>
</tr>
<tr>
<td><strong>DLL</strong></td>
<td>Dynamic Linked Library. A library of functions that an executable may call without being previously linked to the executable.</td>
</tr>
<tr>
<td><strong>FDDI</strong></td>
<td>Fibre Distributed Data Interface. A high-speed (100Mb/s) LAN standard. The underlying medium is fiber optics.</td>
</tr>
<tr>
<td><strong>FEC</strong></td>
<td>Forward Error Correction. A technique for detecting and correcting errors in transmission without requiring retransmission of data by the transmitter.</td>
</tr>
<tr>
<td><strong>FIFO</strong></td>
<td>First in First Out. A common buffer type used to implement queues.</td>
</tr>
<tr>
<td><strong>FTP</strong></td>
<td>File Transfer Protocol. A protocol which allows a user on one host to access, and transfer files to and from, another host over a network.</td>
</tr>
<tr>
<td><strong>GIF</strong></td>
<td>Graphics Interchange Format. A CompuServe™ image format for image files.</td>
</tr>
<tr>
<td><strong>HTML</strong></td>
<td>HyperText Markup Language, web markup language for hyperlinked text. Tags can specify formatting, links to other documents or images.</td>
</tr>
<tr>
<td><strong>HTTP</strong></td>
<td>HyperText Transfer Protocol. An application level protocol for transferring files over the Internet or World Wide Web.</td>
</tr>
<tr>
<td><strong>IDL</strong></td>
<td>Interface Definition Language. The IDL is an implementation-independent language, standardized by the OMG and used to represent an object with its methods and attributes, regardless of platform or language.</td>
</tr>
<tr>
<td><strong>IRC</strong></td>
<td>Internet Relay Chat. A world-wide protocol that allows users to converse with others in real time. IRC is structured as a network of servers, each of which accepts connections from client programs, one per user.</td>
</tr>
<tr>
<td><strong>JAR</strong></td>
<td>Java ARchive. A file format used for aggregating many Java files into one.</td>
</tr>
<tr>
<td><strong>JAX</strong></td>
<td>IBM/Alphaworks jar creating tool that compacts and obfuscates.</td>
</tr>
<tr>
<td><strong>JNI</strong></td>
<td>Java Native Interface. Standard mechanism for connecting Java programs to the native platform through C/C++ code.</td>
</tr>
<tr>
<td><strong>JPG</strong></td>
<td>Image Format for compression of graphic files originally developed by the Joint Photographic Experts Group, a standard for variable level compressed colour images.</td>
</tr>
<tr>
<td><strong>JVM</strong></td>
<td>Java Virtual Machine. The Java Virtual Machine is a strictly defined virtual machine for which an interpreter must be available for the hardware architecture and operating system on which someone will run Java language applications. A JVM is included in most Web-browsers, enabling them to run Java applets.</td>
</tr>
<tr>
<td><strong>LAN</strong></td>
<td>Local Area Network. A data network intended to serve an area of only a few square kilometers or less.</td>
</tr>
<tr>
<td><strong>MIME</strong></td>
<td>Multipurpose Internet Mail Extensions. Classification of content type for use in Internet applications.</td>
</tr>
<tr>
<td><strong>MLE</strong></td>
<td>The maximum likelihood estimate of a parameter from data is the possible value of the parameter for which the chance of observing the data largest.</td>
</tr>
<tr>
<td><strong>MOM</strong></td>
<td>Message Oriented Middleware. Term given to the central router in the final distributed architecture. Also a collective term for software that connects applications running on different systems by sending and receiving application data as messages.</td>
</tr>
<tr>
<td><strong>MTU</strong></td>
<td>Maximum Transmissable Unit. The largest frame length which may be sent on a physical medium.</td>
</tr>
<tr>
<td><strong>NFS</strong></td>
<td>Network File System. A distributed file system protocol suite developed by Sun Microsystems that allows remote file access across a network.</td>
</tr>
<tr>
<td><strong>OMG</strong></td>
<td>The Object Management Group. The OMG is a nonprofit international corporation dedicated to establishing industry guidelines and object management specifications that provide a common framework for distributed application development. See <a href="http://www.omg.org">http://www.omg.org</a>.</td>
</tr>
</tbody>
</table>
ORB  Object Request Broker. Software that allows objects to dynamically discover each other and interact across machines, operating systems and networks.

OSI  Open Systems Interconnection. A suite of protocols, designed by ISO committees, to be the international standard computer network architecture.

Publisher Subscriber  An object oriented design pattern that defines a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

RAPID  The programming language used on the ABB S4 controller.

RMI  Java Remote Method Invocation. The Java RMI enables Java objects to have their methods invoked from Java code running on other virtual machines. It is analogous to a RPC mechanism between Java objects. See http://java.sun.com/products/jdk/rmi/index.html.

RTCP  Real Time Control Protocol. Provides control and quality information on an RTP stream.

RTN  Real Time Network protocol. A RTN protocol is one which guaranties some form of bounded delay and packet loss behaviour.

RTP  Real Time Protocol. Protocol layered on top of UDP for real time services over the internet. RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services.

S4  The name of the ABB controller for the robot manipulator.

SAX  SAX 1.0: The Simple API for XML. See http://www.megginson.com/SAX/index.html.


Sender Receiver  An object oriented design pattern that allows a the sender of event to be decoupled from its receiver, by the use of an abstract interface.

Singleton  An object oriented design pattern that ensures a class only has one instance, and provides a global point of access to it.

TCL  Tool Command Language. A very simple but powerful cross platform scripting language that can also be embedded within application programs.

TCP  Transmission Control Protocol. TCP is layered on top of IP and provides a stream oriented reliable delivery.

UDP  User Datagram Protocol. Datagram protocol used to provide connectionless communication over IP. Lossless delivery is not guaranteed.

URL  Universal Resource Locator. Form in which resources are addressed on the World Wide Web.

UWA  University of Western Australia
XML  eXtensible Markup Language. 2nd generation text web markup language, superset of HTML, see http://www.w3.org/TR/1998/REC-xml-19980210.
APPENDIX A

Related Publications


This appendix describes some of the calculations used in the stereo point algorithm and the augmented reality cursor. In order to translate between 2D images of the workspace and 3D points in the workspace various transformations are required. The transformation from world to camera coordinates is explained in section B.1, this shows how the camera matrix can be built up from certain elemental operations. The transformation back from image coordinates to world coordinates is under-constrained. Two extra constraints are used in the cursor, the plane constraint and the line constraint (section B.2).

## B.1 Elements of the Camera Matrix

The transformation of a point in the world to a point in the camera plane can be split into three separate operations. Firstly the point must be mapped from the world coordinate system to the coordinate system of the camera. This is represented by a homogeneous transformation as shown in equation (B.1). Perspective is applied by multiplying by the perspectivity matrix $\Pi$, and introducing a scaling variable $t$ on the LHS of the equation. Finally a scaling factor to account for the focal length of the camera is applied using scaling matrix $S(f)$. Examination of the resulting matrix (B.4),(B.5) shows that $w$ is always equal to the focal length $f$ and therefore can be omitted from the equation. This leads to a 3 by 4 matrix $C$ as shown in equation (B.6). This maps $x, y$ and $z$ to $u$ and $v$ in the image plane, and this is referred to as the camera matrix.
\[
\begin{align*}
\begin{pmatrix}
x' \\
1
\end{pmatrix} &= \begin{pmatrix}
R & d \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
1
\end{pmatrix}
\quad \Rightarrow x' = w^T x \\
t \begin{pmatrix}
u \\
v \\
w \\
1
\end{pmatrix} &= \begin{pmatrix}
f & 0 & 0 \\
0 & f & 0 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
\quad \Rightarrow t_u = S(f) \Pi_w^T T x \\
t &= \begin{pmatrix}
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
+ d_z \\
t w = f \left( \begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\right) + d_z
\end{align*}
\]

\[
w \equiv f, \quad t = \text{perpendicular distance from image centre to point}
\]

\[
\begin{pmatrix}
u \\
v \\
w \\
1
\end{pmatrix} = \begin{pmatrix}
R & d
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\begin{pmatrix}
1
\end{pmatrix}
\quad \Rightarrow t \equiv \frac{f}{C} x
\]

**B.2 Calculation of Pose Parameters**

Once an element is being dragged, the 2D cursor position must be mapped back to the DOF the cursor represents. In order to perform this mapping the cursor position and an extra constraint are required as an image is only 2 dimensional.

Translation from the 3D workspace model to a 2D image is represented by the affine projection camera equation \( t_u = C x \), where \( C \) is the camera matrix mentioned above, \( x = [x, y, z, 1]^T \) is a homogeneous vector in world coordinates, and \( u = [u, v, 1]^T \) is a homogeneous vector in the image plane. Direct translation from a 2D image point back to 3D point is not possible,
but a 3D image ray from the centre of projection through the 2D image point is given by \( \mathbf{x} = t \mathbf{C}^{-1}_1 \mathbf{u} - \mathbf{C}^{-1}_4 \mathbf{C}_4 \) (the ray equation).

One of two constraints is used to calculate 3D values from the cursor’s position. Either the point is constrained to lie in a predefined plane in the workspace, or on a predefined line; this is illustrated in figure B.1.

**Figure B.1:** Different techniques used to calculate new values of pose. Image rays are either intersected with a plane, or a common perpendicular is used to calculate the closest point on a predefined line.

To translate from screen coordinates to the robot workspace using the plane restriction, the real world point can be calculated by adding the constraining plane equation \( \mathbf{P} \mathbf{x} = 0 \) to the camera equation and solving for the inverse giving (B.7), this is illustrated in figure B.1 part a).

\[
    \begin{pmatrix}
        x \\
        y \\
        z \\
        1
    \end{pmatrix}
    =
    \begin{pmatrix}
        \mathbf{C} \\
        \mathbf{P}
    \end{pmatrix}^{-1}
    \begin{pmatrix}
        u \\
        v \\
        1 \\
        0
    \end{pmatrix}
\]

To translate between screen coordinates and the point on a line in the workspace requires calculation of the common perpendicular of the \((u, v)\) ray and the predefined line, illustrated in figure B.1 part b). If the image ray is written as \( \mathbf{r} = \lambda \mathbf{v}_1 + \mathbf{r}_1 \) (equation B.8) (where \( \mathbf{v}_1 \) is the direction of the image ray and \( \mathbf{r}_1 \) is the projection centre) and the predefined line is written as \( \mathbf{g} = \mu \mathbf{v}_2 + \mathbf{r}_2 \) (equation B.9) then the point on the predefined line in the workspace is given by \( \mu \mathbf{v}_2 + \mathbf{r}_2 \) where \( \mu \) is calculated using equation B.11.
\[ \mathcal{X} = \lambda v_1 + x_1 \quad \text{ray} \quad (B.8) \]
\[ \mathcal{B} = \mu v_2 + x_2 \quad \text{predefined line} \quad (B.9) \]
\[ \lambda v_1 + x_1 - \mu v_2 - x_2 = \nu (v_1 \wedge v_2) \quad (B.10) \]

It can be shown that,
\[ \mu = \frac{((x_2 - x_1) \wedge v_1) \cdot (v_2 \wedge v_1)}{|v_1 \wedge v_2|^2} \quad (B.11) \]

Alternatively to find \( \lambda, \mu \) and \( \nu \) we can solve
\[
\begin{pmatrix}
\lambda \\
\mu \\
\nu
\end{pmatrix}
= \begin{pmatrix}
v_1 & -v_2 & [v_1 \wedge v_2]
\end{pmatrix}^{-1} (x_2 - x_1) \quad (B.12)
\]

Of the different elements that can be changed the HRL’s position is calculated by intersecting the image ray with the base plane. The VRL’s height is calculated using the line method, the current VRL is used as the fixed line. Spin is calculated using a parallel plane to the workspace containing the current GRP. Tilt is computed from intersection with the plane containing the current GRP, the base point, and the spin needle.
Appendix C

Chat Conversations and Comments

This Appendix contains a selection of chat conversations and comments returned from the questionnaire. All conversations and comments refer to the MOM system. The text is reproduced as it was recorded, spelling and typing mistakes have not been corrected.

C.1 Selected Chat conversations

<table>
<thead>
<tr>
<th>Conversation C.1</th>
</tr>
</thead>
</table>
| Typhoid: Hi there  
venkata: Hello there  
Typhoid: Did you see my 5 block stack?  
venkata: No, I didn’t. Can you tell me the x,y coordinates for this  
Typhoid: It goes from x0 y0 to x500 y500 on the board  
Typhoid: about z5 to z300 vertically  
venkata: And where is the stack you just created  
Typhoid: It’s been knocked over. I took a snap-shot - hang on  
venkata: By the way, what’s your geographical location, Typhoid!  
Typhoid: I’m in Jakarta. You?  
venkata: Silicon valley (California) USA  
Typhoid: I’m originally from Perth. I did my EE degree at UWA  
Typhoid: Are you on icq?  
venkata: Nice to know about you. I am original from India and have been in the USA for last ten years.  
venkata: What all you can do these days over the internet, are you surprised too.  
| Typhoid: That url for my stack is hard to remember. I can e-mail or icq you thje url if you are interested.  
venkata: Please do so, I appreciate it.  
Typhoid: What is your addy?  
venkata: By the way, are you involved in the development of this robotic system  
Typhoid: Nope. It’s a mechanical engineering project. I was an electrical engineer  
venkata: Uhh, please stay in touch, my email address is: address removed  
Typhoid: thx - so what is your name?  
venkata: Prasad Iragavarapu  
Typhoid: Nice to meet you Prasad. My name is Bob.  
venkata: It’s nice meeting you. Please stay in touch.  
Typhoid: Sent you an e-mail. Enjoy!  
venkata: Thank you and good bye.  
Typhoid: Hello?  
Typhoid: All yours!  
Typhoid: Sorry gast, you want control back?  

### Conversation C.2

<table>
<thead>
<tr>
<th>mikey</th>
<th>mikey</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnnnnnnoooooooooooooo!!!!!!!!!!!!!!</td>
<td>greater than the roman aqueduct</td>
</tr>
<tr>
<td>what have you done?!?!?!!?</td>
<td>le sigh...</td>
</tr>
<tr>
<td>aaaaaarrrrrrrrrrrrrrgggggghhhhhhhhh...</td>
<td>sniff... sniff... have fun...</td>
</tr>
<tr>
<td>20000916083532637: did i do that? new here</td>
<td>enjoy it while it lasts... for creation is but a fleeting euphoria</td>
</tr>
<tr>
<td>mikey: sniff... sniff... it was so beautiful</td>
<td>mikey: a greater architectural feat than the taj mahal</td>
</tr>
</tbody>
</table>

### Conversation C.3

<table>
<thead>
<tr>
<th>stackman</th>
<th>Iwan</th>
</tr>
</thead>
<tbody>
<tr>
<td>is anyone actually observing this??</td>
<td>Hey Stackman- you ever triep this link?</td>
</tr>
<tr>
<td>Iwan: oops- tried</td>
<td></td>
</tr>
<tr>
<td>stackman: no sure haven’t</td>
<td></td>
</tr>
<tr>
<td>Iwan: You MUST see it- those guys are GODS!</td>
<td></td>
</tr>
<tr>
<td>Iwan: What are you building?</td>
<td></td>
</tr>
<tr>
<td>Iwan: I’m back- no use ignoring me!! What are you building?</td>
<td></td>
</tr>
<tr>
<td>Iwan: Hi! I’m just fine, thanks!</td>
<td></td>
</tr>
<tr>
<td>Iwan: So you are using it for a longer period, already?</td>
<td></td>
</tr>
<tr>
<td>stackman: go ahead! take it!</td>
<td></td>
</tr>
<tr>
<td>Iwan: I gotta work here.... (whenever the Professor looks over my shoulder...)</td>
<td></td>
</tr>
<tr>
<td>Iwan: Problems getting the bricks around with closed gripper??</td>
<td></td>
</tr>
<tr>
<td>Iwan: try gclose,x=271,y=(and so on),gclose!</td>
<td></td>
</tr>
<tr>
<td>Iwan: It works perfectly!</td>
<td></td>
</tr>
<tr>
<td>stackman: well it has been some time since i visited this site the new control is cool!</td>
<td></td>
</tr>
<tr>
<td>Iwan: Where are you from?</td>
<td></td>
</tr>
<tr>
<td>stackman: I’d be in Alaska</td>
<td></td>
</tr>
<tr>
<td>Iwan: Greetings to Alaska! I’m from Germany</td>
<td></td>
</tr>
<tr>
<td>stackman: oops moved before i closed</td>
<td></td>
</tr>
<tr>
<td>Iwan: Happens to me all the time... How did you discover the Robot?</td>
<td></td>
</tr>
<tr>
<td>stackman: searched for robots found this site sounded</td>
<td></td>
</tr>
<tr>
<td>cool (witch of course it was) tried it had fun came back</td>
<td></td>
</tr>
<tr>
<td>afte Iwan: Did you find any other sites that you liked?</td>
<td></td>
</tr>
<tr>
<td>stackman: Can’t say that I did.</td>
<td></td>
</tr>
<tr>
<td>Iwan: You don’t like the coulored Pic, do you?</td>
<td></td>
</tr>
<tr>
<td>stackman: no the color pic is great, it is all the points on the screen that makes the y cam hard to use</td>
<td></td>
</tr>
<tr>
<td>Iwan: Do you know how I can clear the blue &quot;progress report” line??</td>
<td></td>
</tr>
<tr>
<td>Iwan: -um forget abuot it.... found it already</td>
<td></td>
</tr>
<tr>
<td>stackman: yeah click on images above and choose clear path</td>
<td></td>
</tr>
<tr>
<td>Iwan: <em>thanks</em></td>
<td></td>
</tr>
<tr>
<td>stackman: are you going to use this thing or what??</td>
<td></td>
</tr>
<tr>
<td>Iwan: no- I’ll have to work here today, usually I can just doodle around during these lessons, (I’m a Student) but like frenzy!</td>
<td></td>
</tr>
<tr>
<td>Iwan: You see?</td>
<td></td>
</tr>
<tr>
<td>stackman: gee if u just unselect the points they all disappear</td>
<td></td>
</tr>
<tr>
<td>stackman: yahoo i stood one up after 3 hours!</td>
<td></td>
</tr>
<tr>
<td>Iwan: Congratulations- i needed 2 Days to figure it out- (mentally spoken)</td>
<td></td>
</tr>
<tr>
<td>Iwan: But please tell me, how did you deactivate the points??</td>
<td></td>
</tr>
<tr>
<td>stackman: I unselected show points in the images menu</td>
<td></td>
</tr>
<tr>
<td>Iwan: Thats great- I’m just too stupid to try the menu.</td>
<td></td>
</tr>
<tr>
<td>stackman: hey it cleared up my picture a buch it surley helped me so that i could use the cameras</td>
<td></td>
</tr>
<tr>
<td>Iwan: See you stackman - mail me a pic of your creation of the day — gucki address removed</td>
<td></td>
</tr>
</tbody>
</table>
### Conversation C.4

**robobix**: Are we all controlling the bot now?

**Dr. Awesome**: I am whose we?

**robobix**: Hi, how do I control this nice device?

**Dr. Awesome**: Well they say the easiest way is to use the picture and move the cursor

**Dr. Awesome**: Then tell the robot to move to the cursor

**Dr. Awesome**: It takes a little practice though

**robobix**: But how come that you are in command and not I?

**Dr. Awesome**: Click the control robot button, I won’t steal back control this time. And you can take a shot at it.

**robobix**: It don’t move much on the camara?

**Dr. Awesome**: It doesn’t look like it’s been moved at all

**Dr. Awesome**: You can see the black lines on the camera right? Drag them where you want them

**robobix**: So I change this black thing and then press send command?

**Dr. Awesome**: Then right click and select move robot to cursor

**Dr. Awesome**: No send command is just for when you type stuff in like if you want it to move the robot to 250,250,5 then you type x250,y250,z5 and then press send command

**robobix**: Something is happening now 8)

**Dr. Awesome**: Change the quality of YCam back to atleast 10.5 is just plain crappy

**robobix**: I did not get that brik right?

**Dr. Awesome**: The triangles are hard, you can spin the hand though, there’s a little bar on the cursor you can grab that rotates it

**robobix**: This is fun, I will put a link from my page to this

**Dr. Awesome**: Yeah, anyway, I’ve wasted enough time here for one day.

### Conversation C.5

**Kyle**: WAIT!!!!!!

**dopn**: how do you work this

**dopn**: how do you work this

**Kyle**: PLEASE WAIT

**Kyle**: PLEASE WAIT

**Kyle**: ONLY ONE AT A TIME!

**dopn**: shut up

**dopn**: shut up and let me have a go

**Kyle**: could u please wait?

**dopn**: U have been on it 4 ages you are greedy

**Kyle**: IT’S NOT UNCOMMON FOR SOMEONE TO CONTROL FOR 5 HRS. SIR

**dopn**: If you want to control a robot for 5 hours you are a saddo with no life

**dopn**: what have you got to say about that then, KYLE

**Kyle**: PLEASE!! why are u so rude?

**Kyle**: WANT TO USE ???????????

**Kyle**: 200000 YOU WANT TO USE ?????????

**Kyle**: OH WELL..... IM FINISHED

**Kyle**: uhhhhhh

**Kyle**: user 200005118034550 ARE U A “GUEST” ???????

2000051118034550: Is that me? Yes, I am.

**Kyle**: OH...... I WAS WANDERING WAHT ALL THE 3’S WERE 4... U CAN USE NOW..

**Kyle**: OPPS S

2000051118034550: Thanks...
### Conversation C.6

<table>
<thead>
<tr>
<th>Axe:</th>
<th>Jett:</th>
</tr>
</thead>
<tbody>
<tr>
<td>can you set one on top?</td>
<td>ok... watch this</td>
</tr>
<tr>
<td>close, try again</td>
<td>Axe:</td>
</tr>
<tr>
<td>not yet look at zcam</td>
<td>Axe:</td>
</tr>
<tr>
<td>looking good</td>
<td>Axe:</td>
</tr>
<tr>
<td>are you a student there??</td>
<td>Axe:</td>
</tr>
<tr>
<td>no, just found the page</td>
<td>Axe:</td>
</tr>
<tr>
<td>Ohhh thats sucks...knocked right off</td>
<td>Axe:</td>
</tr>
<tr>
<td>I’m still waiting for the pic, but got your msg</td>
<td>Axe:</td>
</tr>
<tr>
<td>just saw what happened</td>
<td>Axe:</td>
</tr>
<tr>
<td>I’m from Vic, using a V90 modem there seems to</td>
<td>Axe:</td>
</tr>
<tr>
<td>be a bit of a delay</td>
<td>Axe:</td>
</tr>
<tr>
<td>Well done</td>
<td>Axe:</td>
</tr>
<tr>
<td>where are you from</td>
<td>Axe:</td>
</tr>
<tr>
<td>Warrnambool Victoria</td>
<td>Axe:</td>
</tr>
<tr>
<td>Canada?</td>
<td>Axe:</td>
</tr>
<tr>
<td>Australia</td>
<td>Axe:</td>
</tr>
<tr>
<td>Ahhhh Sweet.... I am from the states....</td>
<td>Axe:</td>
</tr>
<tr>
<td>You seem to have a fast connection</td>
<td>Axe:</td>
</tr>
<tr>
<td>OC 3...... I work for a internet service Provider</td>
<td>Axe:</td>
</tr>
<tr>
<td>Do you mind zooming out a bit?</td>
<td>Axe:</td>
</tr>
<tr>
<td>I loose the overview</td>
<td>Axe:</td>
</tr>
<tr>
<td>on which cam?</td>
<td>Axe:</td>
</tr>
<tr>
<td>z</td>
<td>Axe:</td>
</tr>
<tr>
<td>Ahhh I got it... watch this</td>
<td>Axe:</td>
</tr>
<tr>
<td>what cam are you using (mainly)</td>
<td>Axe:</td>
</tr>
<tr>
<td>the y cam</td>
<td>Axe:</td>
</tr>
<tr>
<td>whaw</td>
<td>Axe:</td>
</tr>
<tr>
<td>the pics just arrived</td>
<td>Axe:</td>
</tr>
<tr>
<td>These blue trace line got to go..... it makes it messy can't see the target</td>
<td>Axe:</td>
</tr>
<tr>
<td>you can delete them in the images menu</td>
<td>Axe:</td>
</tr>
<tr>
<td>Thanks!!</td>
<td>Axe:</td>
</tr>
<tr>
<td>Here comes the moment of faith.....</td>
<td>Axe:</td>
</tr>
<tr>
<td>well done</td>
<td>Axe:</td>
</tr>
<tr>
<td>WOO HOO got em</td>
<td>Axe:</td>
</tr>
<tr>
<td>use z</td>
<td>Axe:</td>
</tr>
<tr>
<td>use z cam</td>
<td>Axe:</td>
</tr>
<tr>
<td>eeps</td>
<td>Axe:</td>
</tr>
<tr>
<td>looking good</td>
<td>Axe:</td>
</tr>
<tr>
<td>down with it</td>
<td>Axe:</td>
</tr>
<tr>
<td>perfect</td>
<td>Axe: you did it</td>
</tr>
<tr>
<td>Ok its your turn... finish it up.... :-) add that curved pice to the top</td>
<td>Axe:</td>
</tr>
<tr>
<td>i’ve never done this, but I’ll have a go at it</td>
<td>Axe:</td>
</tr>
<tr>
<td>hold up a sec then...</td>
<td>Axe:</td>
</tr>
<tr>
<td>what’s the go?</td>
<td>Axe:</td>
</tr>
<tr>
<td>use the cursor then..... do you see it on the pic</td>
<td>Axe:</td>
</tr>
<tr>
<td>what cursor</td>
<td>Axe:</td>
</tr>
<tr>
<td>let me contrll for a sec.. I will show you the cursour</td>
<td>Axe:</td>
</tr>
<tr>
<td>go</td>
<td>Axe:</td>
</tr>
<tr>
<td>on the menu go to images and then show cursor</td>
<td>Axe:</td>
</tr>
<tr>
<td>ok</td>
<td>Axe:</td>
</tr>
<tr>
<td>do you see it moving when I ajust it?</td>
<td>Axe:</td>
</tr>
<tr>
<td>i see a lot moving, what do you mean?</td>
<td>Axe:</td>
</tr>
<tr>
<td>there is a cross hair.... you have x,y,z tilt,spin</td>
<td>Axe:</td>
</tr>
<tr>
<td>ok</td>
<td>Axe:</td>
</tr>
<tr>
<td>those are the easiest way to contrll it</td>
<td>Axe:</td>
</tr>
<tr>
<td>once you have it in place... right click to control the arm</td>
<td>Axe:</td>
</tr>
<tr>
<td>as a viewer I can’t do much with the cursor</td>
<td>Axe:</td>
</tr>
<tr>
<td>so you give commands and use the blue path to verify then send?</td>
<td>Axe:</td>
</tr>
<tr>
<td>That is not funny... some one is moving the blocks on me</td>
<td>Axe:</td>
</tr>
<tr>
<td>the blue path is to show what you just sent it</td>
<td>Axe:</td>
</tr>
<tr>
<td>it aint me</td>
<td>Axe:</td>
</tr>
<tr>
<td>higher</td>
<td>Axe:</td>
</tr>
<tr>
<td>look at z cam</td>
<td>Axe:</td>
</tr>
<tr>
<td>less y</td>
<td>Axe:</td>
</tr>
<tr>
<td>seems olright</td>
<td>Axe:</td>
</tr>
<tr>
<td>well done</td>
<td>Axe:</td>
</tr>
<tr>
<td>OHHHHHHH NOO</td>
<td>Axe:</td>
</tr>
<tr>
<td>oeps</td>
<td>Axe:</td>
</tr>
<tr>
<td>yeah but check this out</td>
<td>Axe:</td>
</tr>
<tr>
<td>x 5 more</td>
<td>Axe:</td>
</tr>
<tr>
<td>close enough</td>
<td>Axe:</td>
</tr>
<tr>
<td>oooooh</td>
<td>Axe:</td>
</tr>
<tr>
<td>weelll done, thanks for entertaining me buy</td>
<td>Axe:</td>
</tr>
<tr>
<td>laters</td>
<td>Axe:</td>
</tr>
<tr>
<td>sorry i ment bye</td>
<td>Axe:</td>
</tr>
</tbody>
</table>
C.2 Selected Comments

This section consists of a selection of comments made on the Mom system. Comments are included as they were typed, spelling and typing mistakes have not been corrected.

**Comment C.1** Robot seems to move just fine on both the cgi and the java versions. It might just be me, but the pictures seemed to update quicker on the cgi version and that’s why I liked it better than the java. All in all it’s pretty fun to mess with the robot whenever I think about it!

Robert (USA, T1, windows98)

**Comment C.2** I couldn’t connect in and was very disappointed because i have enough power and connection speed to really test it .... sorry

DannyBoy (USA, T3, windows98)

**Comment C.3** i couldn’t figure out how to make the robot move. i tried ”move left” with no results. the hint box was making suggestions like t45 and rz100, i have no idea what that means and although the robot did seem to move when i typed those into the command box, i didnt know what it was doing. it appeared to be moving randomly. eventually i gave for fear i was going to break it.

Fletch (USA)

**Comment C.4** hehe, no I don’t know what a Quaternion is... I love to play with the robot. It’s very cool and the connection was also very well. Is it possible to become a registered user? Keep up the good work! And I will come back :-)

Calimero (The Netherlands, Cable Modem, windows2000)
Comment C.5 The blue history trace lines HAVE to go.... they do nothing but make a mess of
the view... on long winded.. multi move manuvers... the blue line just make it very hard to
see anything. Jett (USA, T3, windows98)

Comment C.6 I wish I could enlarge my camera view frame in the form: I do not want to use
scroll bars any more... tombbombadil (Italy, T1, windows2000)

Comment C.7 Something or one needs to return the blocks to the workspace.... only two blocks
remain Jett (USA, T3, windows98)

Comment C.8 I liked the hints messages - but would like to see a space next to where we type
coord. in that we could select the command like rotate, tilt, open, and so on. This would give
somebody an easier time to select the moves if that person has not worked with an coord.
system or if somebody else was not in. Had a great time and keep it up.
cwr (USA, T3, windows95)

Comment C.9 It would be nice to have some sort of stop function button somewhere. This
could be used when you have entered the wrong command and your structure is about to be
smashed. I dont know if this would work but its a thought... JMK

Comment C.10 I thought the robot was great!I looked at the time after i started and an hour had
gone by.I did not like the annoying tracking line that i had to keep clearing.I also think that
if you change the color of the x,y,z,t lines it would be alot better for the user to tell the
difference between them.All in all it was a great experience and im going to tell my friends
and show my daughter. last thought It was a little slow to respond The only real problem
was almost all the blocks were out of reach. It was GREAT FUN THOUGH!!! rage address
removed rage (USA, 28.8, windows98)

Comment C.11 I have had several student of mine in the 7th and 8th grades wanting me to find
this address. Now I can let them try and move it. The picture is very hard to see, but I think
it is on this end. paul (USA, ISDN, windows95)

Comment C.12 I think that you should have the CGI version available to be used for users behind
firewalls. The integration between the two would be horrendous but it would be very nice
for people behind firewalls and people with slower connections. I do like the GUI of the
Java applet though. Nate (USA, 56.6, windows95)

Comment C.13 cool stuff but very slow on my ancient machine fred (56.6, windows95)

Comment C.14 thanks for the chance to look at something like this, sorry that I do not have the
patience to read and enjoy the site but I did get it to move. brian (Scotland, 56.6, windows98)

Comment C.15 I had an absolute blast with your robot. I hope I didnt damage the end effectors
when I came down on a block. I realized that I was jamming the grippers closed because
they wouldnt open. When I raised the z axis a bit the block flew across the table. Amazing
robot. Would love to have it sitting in my house. Just curious of the cost. Thanks again,
your interface worked great, wish I could see it move at a more real time rate. CPU speed
not helping I’m sure. Thanks again. Henry henry (USA, Cable Modem, windows95)
Comment C.16  Way cool, but still shows how much room there is for better UI to do complex tasks.

It would, for example, be a lot easier to use if there were more scaled live video feeds from the cameras, and each view was surrounded with the controls for the joint shown in the view.

pmb (T3, mac)

Comment C.17  It has been awhile since I visited your page. I like the new control interface with the draggable pointer, It’s really cool! keep up the good work

one more thing- I forgot my password could you help my login name was stackman. I may be able to find it if I could only find the manual I had printed.

Thanks for letting me play with your cool robot.    STACKMAN (USA, 56.6, windows98)

Comment C.18  Excellent robotic control - fast response!!!!!!! Briancoy (USA, 56.6, windows95)

Comment C.19  Why cant we have commands like ”pick object” and a pointer should be provided to the pick the particular object.

ravi (USA, T1, windowsNT)

Comment C.20  some bozo had moved all the blocks outside the workspace so it made it difficult to do much this time. this is probably my 15-20th visit, averaging 20-30 min over a wide range of usage periods (couple of minutes to couple of hours) i have encountered no failure (except for registration) in any of my visits.

spaceman (Canada, Cable Modem, windows98)

Comment C.21  Very interesting to work (play) with. Modem connection is too slow. I will try again next week, when my satellite dish (400kbps+) is back up

tinman (USA, 56.6, windows98)

Comment C.22  I am a 2nd year student at the University of Melbourne studying Engineering(Mechatronics)/Computer Science. Being a second year student I haven’t had too many opportunities to experiment on actual hardware, so I found the experience quite interesting. The interface is quite straight forward and the robots real world perception (ie realising that the given command will cause it to strike something) is excellent.

mick (Australia, 56.6, windows98)

Comment C.23  A quaternion is exactly one fourth a wholernion

Saggy (England, 56.6, windows98)

Comment C.24  Most excellent. But the right click menu’s don’t appear to be working with Netscape Communicator 4.7

crispoid (Australia, Cable Modem, windows2000)

Comment C.25  I work in ABB Robotics, so I found it fascinating to be able to move an ABB robot being so far away. The delay is still big so it is very difficult to make a large program but moving step by step works fine. Do you have the vision system connected to the robots controller or is it independant? Is it an S4C or S4C Plus controller? This was the first time I ever used Telerobot so I haven’t used the CGI version, I think that it worked fine with Java. It is sometimes difficult to see the position of the robot with regard to a certain piece, because of the perspective of the cameras, I actually crashed once. Maybe you should install the software CDPR (collision detection with path retraction) so if someone crashes, the robot will move a certain distance in +Z.

I had a lot of fun playing with this robot, i’ll do it more often.

Regards

Ulrik    IRB140 (Mexico, 56.6, windows98)
Comment C.26 I used Netscape with Java Plug-in. It works great, I wish the images were a little bigger. Good work, I like the Java Interface and the chatting with others... the Hint books is nice, but to often...

keep doing the good work... jya2 (USA, Cable Modem, windows98)

Comment C.27 It was fun. What a unique experience. Thanks for sharing!

P.S.-I left the word, "HI," on the board (in blocks of course).

Waber342 (USA, Cable Modem, windows98)

Comment C.28 This is wonderfull. I am bookmarking this site and tonight when I go to class at the college, I am going to tell everyone!

I started out with the numbers, (I do 3d animations, so x,y, and z are friends of mine), but the cursor control is so much more fun!!

I made a stack of junk... I feel so proud.

NICE work, fellas!

Lindsey, Ventura, Ca. USA Pixel (USA, ADSL, windowsNT)

Comment C.29 The robot moved OK. Commands were sent and executed correctly, and video feedback delay was approximately 2-3 secs.

The only problem was the graphical control interface, with the cursor moves, since it is not easy to interpret the position of the cursor in 3D with respect to the robot manipulator and the objects.

Besides this inconveniency, the system worked fine. ktzaf (Greece, T1, windows2000)

Comment C.30 I think it great. I loved it and so did my 10 year old son.

shortwave (USA, ADSL, windows98)

Comment C.31 The telerobot is great!! I would like to see faster updates, but that is a problem on my end. (modem not fast enough?) I am a student in the robotics program at a local college. It would be neat if we could set up a system here and have the two play chess or something. Please keep the project going. Thanks Rod aka bigdogrod

bigdogrod (USA, 33.3, windows98)

Comment C.32 Works much better with Satellite Conection.Last visit I used 56k, was too slow. feed back is still little slow. But I was able to pick-up and move 2 blocks. I will try again later.

tinman (Satellite Modem, windows98)

Comment C.33 Haven’t done it in over a year. Love the color! and the new easier commands. Wish he’d opened the gripper and not wrecked my 4 stack. dozamd (USA, ADSL, mac)

Comment C.34 Interesting but it’d take time to achieve things. The commands executed within a few seconds but then it took another few seconds for the new picture to come.

r (Uk, windows98)

Comment C.35 Nice interface, perhaps you should add a way to measure angles btwn points too, or did I miss something? jjonny (Sweden, T1, windowsNT)
**Comment C.36** the font size of your java applet is too small for my 17 inch screen with the settings i need to see the live video ,etc... could you include scalable fonts so we don’t have to read very very tiny letters ?

**2000110923184924 (Cable Modem)**

**Comment C.37** Awesome interface - I really enjoyed using it to stack blocks on top of each other. The large Java application worked without any glitches or crashes. This is an incredible project, and you should be proud of your achievement. If there’s a way I can do more testing, please let me know.

Stephen Borg  
*TheBorg (Canada, Cable Modem, windows2000)*

**Comment C.38** i really enjoyed my visit to the telerobot site i tried to chat with others but when your trying to move the robot the last thing your thinking about is chatting also there has to be an allotment of time for each user i think that would have made it more enjoyable i didnt control it for too long because i wanted everyone to enjoy the robot as much as i did maybe tasks that are not so monumental and that are a bit simpler would get more people interested in it instead of watching someone there for hours building a monument

*drunk (United states of america, ADSL, windowsNT)*