

# Programming the SIMPL Way

John Collins      Robert Findlay

Copyright ©2010 by iCanProgram Inc.

The rights of John Collins and Robert Findlay to be identified as authors of this material in accordance with the Copyright, Designs and Patents Act.

ISBN 978-0-557-47131-7

The authors have taken reasonable care in the preparation of this document and make no expressed or implied warranty of any kind and assume no responsibility for errors and omissions. No liability is assumed in connection with or arising from the use of the information or programs contained within this document.

*Dedicated to Dan Hildebrand*

A deep thinker and a really nice guy who was taken from us too soon.



# Contents

Preface - Third Edition	vii
Preface - Second Edition	ix
Preface - First Edition	xi
<b>I SIMPL Beginnings</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Beginnings . . . . .	3
1.2 Summary . . . . .	6
1.3 Next Chapter . . . . .	6
<b>2 A SIMPL Story</b>	<b>7</b>
2.1 The Story of SIMPL . . . . .	7
2.2 SIMPL Tales . . . . .	11
2.3 About Programming Paradigms . . . . .	13
2.4 Summary . . . . .	15
2.5 Next Chapter . . . . .	15
<b>II SIMPL Elements</b>	<b>17</b>
<b>3 SoftwareICs: Theory</b>	<b>19</b>
3.1 SoftwareICs . . . . .	19
3.2 SIMPL Axioms . . . . .	22
3.3 SIMPL as a SoftwareIC Toolkit . . . . .	24
3.4 Summary . . . . .	25
3.5 Next Chapter . . . . .	26

<b>4 SIMPL Core</b>	<b>27</b>
4.1 The Core Elements of SIMPL . . . . .	27
4.2 SIMPL Programming Languages . . . . .	28
4.3 Local SIMPL . . . . .	28
4.3.1 SIMPL Names . . . . .	29
4.3.2 SIMPL Messaging . . . . .	29
4.4 SRR Example . . . . .	34
4.4.1 The Receivers . . . . .	35
4.4.2 The Senders . . . . .	42
4.4.3 What's In a Name? . . . . .	49
4.5 Return Codes . . . . .	50
4.5.1 Warnings and Errors . . . . .	51
4.6 Security . . . . .	51
4.7 Installing SIMPL . . . . .	52
4.8 Summary . . . . .	52
4.9 Next Chapter . . . . .	52
<b>5 Messages</b>	<b>53</b>
5.1 Messages and Messaging . . . . .	53
5.2 Messages . . . . .	54
5.3 Using Messages . . . . .	56
5.3.1 Packing and Unpacking Messages . . . . .	56
5.3.2 Tokenized Binary Messages . . . . .	61
5.4 Summary . . . . .	76
5.5 Next Chapter . . . . .	76
<b>6 SoftwareICs: Practice</b>	<b>77</b>
6.1 Applied SoftwareICs . . . . .	77
6.2 Simulator . . . . .	78
6.3 Stimulator . . . . .	79
6.4 Relay . . . . .	80
6.5 Proxy . . . . .	82
6.6 Stack . . . . .	85
6.7 Courier . . . . .	87
6.8 Agency . . . . .	88
6.9 Broadcaster . . . . .	91
6.10 Scheduler . . . . .	93
6.11 Statemachine . . . . .	94
6.12 Polling Emitter . . . . .	97

6.13	Summary	99
6.14	Next Chapter	99
<b>7</b>	<b>Surrogates and Networking</b>	<b>101</b>
7.1	Background	101
7.2	Surrogates	102
7.3	Remote Name Location	104
7.4	Protocol Router	106
7.4.1	Name Locate Operations	107
7.5	Surrogates and Protocols	107
7.5.1	TCP/IP Surrogate	108
7.5.2	RS-232 Surrogate	108
7.5.3	tclSurrogate	109
7.5.4	QNX Surrogates	111
7.5.5	Internal Parameters	111
7.5.6	Endian Issues	111
7.6	Summary	113
7.7	Next Chapter	114
<b>8</b>	<b>Raspberry Pi and Arduino</b>	<b>115</b>
8.1	Raspberry Pi	115
8.2	Arduino	116
8.3	Hangman Game	117
8.3.1	Hangman Design	117
8.3.2	SIMPL Receiver	119
8.3.3	SIMPL Sender	121
8.3.4	What you should see on the Development Host	130
8.4	Summary	131
8.5	Next Chapter	131
<b>III</b>	<b>SIMPL Toolkit</b>	<b>133</b>
<b>9</b>	<b>Testing Framework</b>	<b>135</b>
9.1	What is a Testing Framework?	135
9.2	Summary	146
9.3	Next Chapter	146

<b>10 SIMPL Sandbox</b>	<b>147</b>
10.1 Sandboxes for Testing . . . . .	147
10.2 Summary . . . . .	150
10.3 Next Chapter . . . . .	150
 <b>IV SIMPL Example Project</b>	 <b>151</b>
<b>11 A SIMPL Project - Putting It All Together</b>	<b>153</b>
11.1 Project Specification . . . . .	153
11.2 The Design Meeting . . . . .	154
11.3 The First Cut . . . . .	155
11.4 Sudoku . . . . .	157
11.5 Summary . . . . .	158
11.6 Next Chapter . . . . .	158
 <b>12 Test Plan</b>	 <b>159</b>
12.1 Sudoku Test Plan . . . . .	159
12.2 The Unit Tests . . . . .	161
12.3 The Integration Tests . . . . .	162
12.4 Summary . . . . .	162
12.5 Next Chapter . . . . .	163
 <b>13 Directory Tree</b>	 <b>165</b>
13.1 Good Housekeeping . . . . .	165
13.2 Seeding the Code . . . . .	166
13.3 Seeding the Tests . . . . .	167
13.4 Summary . . . . .	168
13.5 Next Chapter . . . . .	168
 <b>14 Sims and Stims</b>	 <b>169</b>
14.1 Unit Testing . . . . .	169
14.2 What sort of Messages Do We Need? . . . . .	170
14.3 Global Variables . . . . .	172
14.4 Simulator and Stimulator . . . . .	172
14.5 Simulator Logic . . . . .	173
14.6 Stimulator Logic . . . . .	179
14.7 Testing Sim and Stim . . . . .	189
14.8 Summary . . . . .	192

---

14.9 Next Chapter . . . . .	192
<b>15 Sudoku Engine Program</b>	<b>193</b>
15.1 Building the Engine Program . . . . .	193
15.2 Testing the Engine . . . . .	200
15.3 Self Test . . . . .	203
15.4 Summary . . . . .	203
15.5 Next Chapter . . . . .	203
<b>16 Python/Tk GUI</b>	<b>205</b>
16.1 The Python GUI . . . . .	205
16.2 Building the Python/Tk GUI . . . . .	206
16.3 Testing the Python/Tk GUI . . . . .	211
16.4 Summary . . . . .	214
16.5 Next Chapter . . . . .	214
<b>17 Integration Testing - Python GUI/ Sudoku Engine</b>	<b>215</b>
17.1 Python GUI Integration Test . . . . .	215
17.2 Summary . . . . .	218
17.3 Next Chapter . . . . .	218
<b>18 Sudoku Evolution</b>	<b>219</b>
18.1 Forking Engine . . . . .	219
18.2 Networked Sudoku . . . . .	227
18.3 Tcl/Tk GUI . . . . .	231
18.4 Java/Swing GUI . . . . .	240
18.5 CGI . . . . .	251
18.5.1 Web Server . . . . .	251
18.5.2 Sudoku via Web Browser . . . . .	252
18.5.3 HTML/CSS . . . . .	253
18.5.4 CGI Script . . . . .	258
18.5.5 Possible Problems . . . . .	262
18.5.6 In the Real World . . . . .	263
18.6 Summary . . . . .	263
<b>19 Conclusion</b>	<b>265</b>

<b>V Appendices</b>	<b>267</b>
<b>A Installation</b>	<b>269</b>
A.1 Installing SIMPL Core Code . . . . .	269
A.2 Building the SIMPL Core Libraries . . . . .	270
A.3 SIMPL Public SoftwareICs Code Repository . . . . .	272
A.4 SIMPL Testing Framework . . . . .	273
A.5 Python SIMPL . . . . .	274
A.6 Tcl SIMPL . . . . .	275
<b>B Language Packages Summary</b>	<b>277</b>
<b>C C Library Functions</b>	<b>279</b>
C.1 Function Synopsis . . . . .	279
C.2 Function Summary Table . . . . .	301
<b>D Arduino Library Functions</b>	<b>303</b>
D.1 Function Synopsis . . . . .	303
D.2 Function Summary Table . . . . .	317
<b>E Python Module Functionality</b>	<b>319</b>
E.1 Libraries and Modules . . . . .	319
E.1.1 csimpl/simpl2.so/simpl3.so . . . . .	319
E.1.2 tclsimpl . . . . .	320
E.1.3 Module Summary Table . . . . .	320
E.2 Main Functionality . . . . .	321
E.2.1 Main Function Synopsis . . . . .	321
E.2.2 Main Function Summary Table . . . . .	335
E.3 Packing and Unpacking Functions . . . . .	335
E.3.1 Packing/Unpacking Function Synopsis . . . . .	335
E.3.2 Packing Functions . . . . .	336
E.3.3 Unpacking Functions . . . . .	336
E.3.4 Example 1 . . . . .	337
E.3.5 Example 2 . . . . .	342
E.3.6 Packing/Unpacking Function Summary Table . . . . .	346
E.4 Miscellaneous Functions . . . . .	346
E.4.1 Messages without Packing and Unpacking . . . . .	346
E.4.2 Example 1 . . . . .	348
E.4.3 Example 2 . . . . .	351

E.4.4	Miscellaneous Function Summary Table	355
<b>F</b>	<b>Java Language Methods</b>	<b>357</b>
F.1	SIMPL Classes	357
F.1.1	Jsimpl	357
F.1.2	Ssimpl	357
F.1.3	Class Summary Table	358
F.2	Main Method Synopsis	358
F.2.1	Main Method Summary Table	372
F.3	Packing/Unpacking Method Synopsis	372
F.3.1	Packing Methods	373
F.3.2	Unpacking Methods	373
F.3.3	Packing/Unpacking Method Summary Table	374
F.3.4	Example	375
F.4	Miscellaneous Methods	382
F.4.1	Miscellaneous Method Summary Table	383
<b>G</b>	<b>Tcl Library Functions</b>	<b>385</b>
G.1	Procedure Synopsis	385
G.2	Procedure Summary Table	391
<b>H</b>	<b>TCP/IP Surrogate - Details</b>	<b>393</b>
H.1	Operation	393
H.2	Starting Up	395
<b>I</b>	<b>RS-232 Surrogate - Details</b>	<b>397</b>
I.1	Operation	397
I.2	Starting Up	399
<b>J</b>	<b>tclSurrogate Protocol</b>	<b>401</b>
J.1	General Message Format	401
J.2	Some Terminology	402
J.3	Tokens	402
J.4	NAME_ATTACH	402
J.5	NAME_DETACH	403
J.6	NAME_LOCATE	403
J.7	SEND	403
J.8	REPLY	404
J.9	RELAY	404

J.10 IS_LOGGER_UP . . . . .	404
J.11 LOGIT . . . . .	404
J.12 SEND_NO_REPLY . . . . .	405
J.13 ACK . . . . .	405
J.14 PING . . . . .	405
<b>K Surrogate Internal Messages</b>	<b>407</b>
<b>L Apache Configuration</b>	<b>409</b>
<b>M Trace Logger</b>	<b>411</b>
<b>N Advanced SIMPL for C Programmers</b>	<b>415</b>
N.1 Background . . . . .	416
N.2 Sample Code . . . . .	416
<b>O Warning and Error Messages</b>	<b>421</b>
<b>P SIMPL License</b>	<b>423</b>
<b>Q SRY Kernel Module</b>	<b>425</b>
Q.1 Introduction . . . . .	425
Q.2 Background . . . . .	426
Q.3 SRY Organization . . . . .	427
Q.3.1 User Space Applications . . . . .	428
Q.3.2 User Space SRY Library . . . . .	437
Q.3.3 Kernel Space SRY Module . . . . .	438
Q.4 Starting SRY . . . . .	444
Q.5 SRY Development Directory Structure . . . . .	446
Q.5.1 Makefile . . . . .	446
Q.5.2 Kernel Module Development Code . . . . .	447
Q.5.3 C/C++ library Development Code . . . . .	447
Q.5.4 Include . . . . .	448
Q.5.5 Libraries . . . . .	448
Q.5.6 Binaries . . . . .	448
Q.5.7 Shell Scripts . . . . .	448
Q.5.8 Tools . . . . .	448
Q.5.9 Testing . . . . .	449
Q.5.10 TCP/IP Surrogate . . . . .	450
Q.5.11 Python SRY . . . . .	453

Q.5.12	Python Modules . . . . .	456
Q.5.13	Benchmarking . . . . .	456
Q.6	Example . . . . .	456
Q.6.1	Fractals and the Example Programs . . . . .	457
Q.6.2	C/C++/Xwindows Receiver . . . . .	461
Q.7	Summary . . . . .	489
Q.8	Installing SRY . . . . .	489
<b>R</b>	<b>Glossary of Terms</b>	<b>491</b>



# List of Figures

1	SIMPL Thinking . . . . .	xii
4.1	Send/Receive/Reply . . . . .	30
5.1	SIMPL Tokenized Message . . . . .	61
6.1	Simulator . . . . .	78
6.2	Stimulator . . . . .	79
6.3	Relay . . . . .	81
6.4	Proxy . . . . .	83
6.5	Stack . . . . .	85
6.6	Courier . . . . .	87
6.7	Agency . . . . .	89
6.8	Broadcaster . . . . .	91
6.9	Scheduler . . . . .	93
6.10	Door . . . . .	95
6.11	Statemachine . . . . .	96
6.12	Polling Emitter . . . . .	98
7.1	Surrogate Programs . . . . .	104
7.2	tclSurrogate . . . . .	110
8.1	Hangman Game . . . . .	117
8.2	Game Network . . . . .	118
8.3	Hangman Screen . . . . .	130
9.1	SIMPL Test Framework . . . . .	138
10.1	SIMPL Sandbox . . . . .	149
11.1	System Picture . . . . .	154

11.2 Sudoku Subsystem Picture . . . . .	156
11.3 A Typical 9x9 Sudoku Puzzle . . . . .	157
12.1 sb000 - sim/stim unit test . . . . .	160
12.2 sb001 - Sudoku engine unit test . . . . .	160
12.3 sb002 - Python GUI unit test . . . . .	160
12.4 sb002s - Non-Linux Python GUI unit test . . . . .	161
12.5 sb004 - Python GUI/Sudoku engine test . . . . .	161
14.1 Sim/Stim SIMPL Test . . . . .	190
15.1 Engine Unit Test . . . . .	201
16.1 Python/Tk Unit Test . . . . .	211
16.2 Python/Tk GUI screen . . . . .	214
17.1 Python Integration test . . . . .	215
18.1 Forked Engine . . . . .	220
18.2 Networked Sudoku . . . . .	227
18.3 sb003 - Tcl GUI unit test . . . . .	232
18.4 sb003s - Non-Linux Tcl GUI unit test . . . . .	232
18.5 sb005 - Tcl GUI/Sudoku engine test . . . . .	233
18.6 Tcl/Tk gui screen . . . . .	236
18.7 sb007 - Java/Swing test . . . . .	240
18.8 Java/Swing gui screen . . . . .	250
18.9 sb006 - Python-CGI test . . . . .	251
18.10HTML gui screen . . . . .	258
H.1 TCP/IP Surrogate Networking . . . . .	394
I.1 RS-232 Surrogate Networking . . . . .	398
M.1 trace logger . . . . .	412
Q.1 SRY Breakdown . . . . .	428
Q.2 SRY Python . . . . .	454
Q.3 Menu One . . . . .	483
Q.4 Menu Two . . . . .	483
Q.5 Menu Three . . . . .	484
Q.6 Mandelbrot Fractal . . . . .	485
Q.7 Julia Fractal . . . . .	486

Q.8 Cosine Fractal . . . . .	486
Q.9 Exponential Fractal . . . . .	487
Q.10 Barnsley Fractal . . . . .	487
Q.11 Legendre Fractal . . . . .	488
Q.12 Sierpinski Fractal . . . . .	488



# List of Tables

4.1 Telephone/SIMPL Analogy . . . . .	29
7.1 Endian Memory Storage of 0xABCD . . . . .	112
7.2 Binary $\Leftrightarrow$ Character Conversion Functions . . . . .	113
B.1 Summary of SIMPL Language Packages and Compatibility . . . . .	277
C.1 Summary of ‘C’ Functions . . . . .	301
D.1 Summary of ‘C’ Functions for Arduino . . . . .	317
E.1 Summary of Python Modules . . . . .	320
E.2 Summary of Python Main SIMPL Functions . . . . .	335
E.3 Python SIMPL Packing/Unpacking Functions . . . . .	347
E.4 Summary of Python SIMPL Miscellaneous Functions . . . . .	355
F.1 Summary of Java Classes . . . . .	358
F.2 Summary of Jsimpl/Ssimpl Methods . . . . .	372
F.3 Java Packing/Unpacking Methods . . . . .	374
F.4 Java Miscellaneous Methods . . . . .	383
G.1 Summary of Tcl Procedures . . . . .	391
K.1 Surrogate Tokens . . . . .	408
O.1 Warning and Error Messages . . . . .	422
Q.1 C++ SRY Class Member Functions . . . . .	429
Q.2 C SRY Functions . . . . .	429
Q.3 Library Call - System Call Correspondence . . . . .	438
Q.4 System Call - File Operations Method Correspondence . . . . .	439
Q.5 SRY Library/Module Calls via ioctl . . . . .	440

Q.6 SRY Development Directory . . . . .	446
Q.7 Local Test Programs . . . . .	450
Q.8 Network Test Programs . . . . .	450
Q.9 TCP/IP Surrogate Command Line Arguments . . . . .	451
Q.10 Python Sen and Rec Class Methods . . . . .	454
Q.11 Python Msg Class Methods . . . . .	455

# Preface - Third Edition

SIMPL continues to be used in lots of projects. This edition discusses SIMPL on the Raspberry Pi embedded Linux processor.<sup>1</sup>

While strictly not a full SIMPL implementation, we also cover the use of SIMPL to facilitate Arduino projects.<sup>2</sup>

SIMPL has been ported to the Cygwin POSIX implementation for Windows OS.<sup>3</sup> However, due to internal implementation of named pipes in Cygwin while the SIMPL code could compile and run the performance was orders of magnitude slower than on a Linux OS.

Since the second edition, the Python SIMPL infrastructure has undergone significant revisions.

## Acknowledgements

We would like to thank all the users of SIMPL who continue to inspire us to continuously improve the codebase.

---

<sup>1</sup>Raspberry Pi online:

<http://www.raspberrypi.org>.

<sup>2</sup>The Arduino project online:

<http://www.arduino.cc>.

<sup>3</sup>Cygwin online:

<https://www.cygwin.com>.



# Preface - Second Edition

SIMPL is an active open source project. While the core SIMPL library has been stable for some time, work continues on extending the reach of SIMPL to other programming languages and platforms. Since the first edition was published, Java-SIMPL has been extended with a full JNI library. The Java-SIMPL for non-Linux OSs (via tclSurrogate) was completely rewritten as well.

The Python-SIMPL suite has been enhanced to include support for SIMPL as a form of **web services light** where the principal human interface to the SIMPL application is a web browser. There have been several enhancements to the Python-SIMPL library suite as well.

While much of the book is relatively unchanged, aside from minor corrections, a number of programming language appendices have been added to support the growth of SIMPL in Python and Java. The Sudoku project part of the book has been expanded to accommodate the greater reach of SIMPL. The chapter on tokenized messaging has been absorbed into and replaced by a new chapter which discusses messages more generally and extensively.

SIMPL has always been available as a source code tarball which had to be manually installed. While we have yet to achieve a breakthrough into the standard package management systems (Debian, RPM), the SIMPL build tree has undergone a complete makeover to be more compatible with such systems. In addition to the source tarball, SIMPL users can now avail themselves of self-installing archives in both source and binary format. This greatly simplifies the entry for new SIMPL developers and users. The sample code associated with this book is now available in these formats as well.

## Acknowledgements

On this, the second edition of the book, we would like to thank Steve Meyer for his infectious enthusiasm and many solid technical suggestions with respect to improvements and new directions for SIMPL.



# Preface - First Edition

The theme of this book is really the story of a software journey. Because this journey has been undertaken by real people, we hope that it serves to put a human face on the subject. This a book about SIMPL. SIMPL, at its most basic level is a library of functions that enable two or more software programs to communicate with each other by sending and receiving information in the form of enclosed messages. SIMPL modules can be many things. They are great containers for encapsulating complexity in a software algorithm. SIMPL modules are both network and programming language agnostic within its supported libraries. However, the story of SIMPL is more about the continuous and pragmatic refinement of our software development toolkit under the open source framework. This book is our attempt to wrap that experience in a package so that others can learn, adopt, and carry on with the journey.

Over the years we have noticed a scarcity of mentoring and apprenticeship in the programming world. Everyone we know has learned pretty much on their own by experimentation or from observing and adopting the style and code of colleagues during a project. The idea of learning directly from a more seasoned person has all but disappeared; there never seems to be time for teaching. The project simply has to get done and out there. Experts are harder to come by because new software tools are coming along with ever increasing frequency. By the time someone has learned enough to be competent, the tool is considered *old* and needs to be *replaced* by the flavour of the month and everyone starts the learning curve again.

In a very real sense, the SIMPL toolbox will never get old and out of date because it is more than a library. SIMPL is also a viewpoint that reinforces modularity and encourages order and structure, all timeless practices for good software developers.

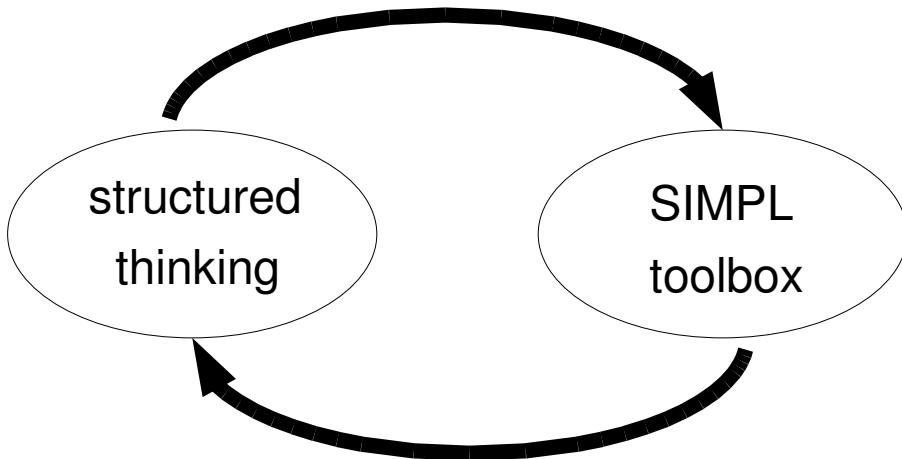


Figure 1: SIMPL Thinking

## Who is this book for?

Who is the target audience? A very good question and we wish we had very good answer. Generally speaking, software developers open to new approaches for improving the efficiency and organization of their complex projects will get the most from this book. Advanced programmers should have no difficulty understanding the technical aspects of the book whereas less experienced developers may have to work a little harder. Experience teaches programmers how to recognize patterns and map those from toolkit to project. Experienced SIMPL developers become very adept at discovering software patterns and mapping them into the SIMPL toolkit. However, experienced programmers can become set in their ways. Less entrenched developers will be more open to the SIMPL approaches to software design, structure, testing and project management. While the book discusses various programming languages ('C', Tcl, Python, and Java) and constructs in those languages, it is not a book about programming languages. It is a book about a toolkit and a method for decomposing and handling complex software problems. As such, software project managers and software quality assurance workers could enjoy and learn much from this book. Furthermore, SIMPL is one

of many thousands of small open source projects and its story might appeal to others working on similar open source projects.

## Useful Background

The SIMPL toolkit and many of the available examples are coded in ‘C’. Every effort has been made to keep the ‘C’ code snips which appear in the book straightforward and heavily commented, but for a full appreciation some knowledge of ‘C’ is necessary. SIMPL modules are not restricted to the language of the SIMPL library. Tcl/Tk, Python/Tk, and Java/Swing have been used to illustrate examples in the book as well. Once again, while these code snips are kept straightforward, some background in any of these languages will enrich your experience with this material. There is a lot of SIMPL sample code available online including all the code for the project described in the book. For the most complete experience, access to a Linux environment to run these would be helpful. SIMPL was developed for Linux and it is still most at home there.

If you lack any of the above, don’t be overly concerned. The story of SIMPL woven into the fabric of this book means that you can still take away many useful ideas.

## SIMPL Dilemma

As authors we face a dilemma. How do we illustrate SIMPL in a clear and concise manner when the SIMPL toolkit excels at solving complex problems? If we *dumb down* our example scope we run the risk of leaving the reader with the impression that SIMPL is an overly complex tool to use. Yet, if we illustrate with sufficiently realistic scope we run the risk of losing our point amongst the details of a complex problem.

Our reader must look beyond the simple number of lines of code metric to appreciate the SIMPL examples we use. To fully appreciate the power of SIMPL one must understand its power in the problem decomposition, testing and maintenance phases of a non-trivial project. The *Hello World* type examples are not complex enough to have significant issues with problem decomposition, testing or maintenance. When faced with our illustrative examples we think it helpful to ask some scope expanding *what if* questions such as:

- What if this example were to be run across a network? What if that network consisted of different operating systems on each node?

- What if this was just the first cut in a more complex algorithm? How would the code evolve to an ever more complex feature set? How would you efficiently test this new feature set?
- What if this code required a team of developers to create it in a timely manner? What if that team was geographically dispersed?

There is a sample project contained within the fourth part of the book which faces the SIMPL dilemma. We have attempted to strike a balance between conciseness of explanations and details of scope. Despite this we hope that it will give the reader the essential ideas that lie behind all SIMPL projects: large and small.

## Note on Library References

The SIMPL toolkit contains libraries of functions/procedures etc. References to these are made throughout the text and appear as ***emboldened italics*** in order to reinforce their meaning.

## Acknowledgements

We would like to take this opportunity to thank all of those people who have worked to make Linux, LaTeX et al. useful and available. Like SIMPL, they are members of the open software movement which we strongly endorse.

We would also like to thank Mohammed El-Korazati, Peter Hiscocks, Graham McCormack, and Peter Spasov for their many valuable and insightful comments on the original manuscript.

# Part I

# SIMPL Beginnings



# Chapter 1

## Introduction

### 1.1 Beginnings

To understand what SIMPL (Synchronous Interprocess Messaging Project for Linux)<sup>1</sup> programming is, it is important to step back a bit and ask what software is really all about. Software is what enables hardware to come alive and function in a purposeful way. The art of writing software is similar in many ways to creating anything else, like creating music. Think of hardware as the musical instruments. The software tools such as programming languages, compilers and SIMPL are like musical notes and techniques. You have to learn those first. Having mastered the fundamentals, you can learn to play a few songs that others have written. With lots of practice, you can begin to play those well enough to begin to add your own interpretations. Eventually you get the urge to create entirely new songs of your own. You write a bit, play it back, adjust and eventually arrive at your new musical creation.

Not all software tools allow you to *write a bit, play it back, adjust*. The SIMPL toolset not only allows that approach, it encourages it.

What exactly is SIMPL? SIMPL is really two things. Firstly, SIMPL is an LGPL'd open source library that you can dynamically (or statically) link your code to. Secondly, SIMPL represents a very good paradigm for designing software. It promotes encapsulation of complexity, modularization, testability, and ready extensibility; all good things in a software package.

Much of the software we are familiar with is either so-called shrink wrapped software (eg. Linux distributions or Open Office) or is associated with the Internet (eg. browsers or firewalls). This however, is not where the bulk of software

---

<sup>1</sup>Don't worry, the name is historic; SIMPL runs on other platforms as well.

development activity occurs. The largest software markets are associated with the software we rarely see, such as banking systems or embedded software controlling our car engine parameters. Software tools and methodologies evolve as attempts are made to improve the efficiency of the production of this commercial custom software.

Historically, computers themselves were expensive to procure and to operate. In that environment it made sense to apply software methodologies which emphasized as much design and forethought as possible before someone actually sat down to commit it to code and then run it. Large civil engineering projects are like this; you want to do all the design and stress calculations upfront before you start pouring the concrete.

Cracks started to appear in these front-loaded methodologies as computers became less and less expensive per unit of computational power. With modern PCs the cost of running a computer program is vanishingly small. Most of the software development costs are now people costs. With the advent of the Internet and open source tools the cost of entry into software development is now becoming vanishingly small as well. Couple this computer hardware and toolset price point shift with a marketplace which demands increasingly short times to market, and you start to need a radically new software development methodology in order to remain in the game. SIMPL is one such methodology.

A desirable feature of any software toolbox is its ability to encapsulate complexity inherent in many software problems. Object oriented languages use objects to encapsulate functionality. It can be argued that in some cases they also work well to encapsulate complexity. However, in many other cases OOP languages do little to encapsulate the inherent complexity associated with a given software problem. Furthermore, object oriented languages are neither necessarily simple to understand nor to master. One of our aims will be to show that encapsulation of complexity can be done simply.

We will employ the concept of a software IC to illustrate the encapsulation of complexity. Here we will draw on the experience of hardware designers faced with similar complexity issues. It wasn't that many years ago when electronic circuits were built up of discrete components such as transistors, resistors, capacitors, and so on. As complexity of electronic circuits increased it became impossible for hardware engineers to manage this complexity. Enter the integrated circuit (IC). What this did for the hardware engineer was to take functionality which would have previously been provided by a custom circuit board with discrete components and replace it by a silicon chip with well-defined behavioural characteristics and pin outs. This innovation is what has allowed the hardware engineers to make the spectacular progress we have witnessed in the past quarter

century.

Software designers never had that equivalent encapsulating technology. Fortunately, modern multitasking operating systems provide us with a ready analogue to the hardware integrated circuit, namely a process running in a protected mode. SIMPL messaging provides the analogue to the IC pins. With SIMPL and a basic language such as ‘C’ we are going to show that we can achieve something remarkably close to a software IC without having to resort to a complex language or hard to read source code.

Once you have the SIMPL library installed, you can write your modular code such that it readily exchanges Send/Receive/Reply style messages, hence the M-Messaging in the name SIMPL. SIMPL comes with a rich, yet simple set of functions for Interprocess Communication (IPC) (the I-Interprocess in the name). These functions include ***name\_attach***, ***name\_locate***, ***Send***, ***Receive***, ***Reply*** and a few lesser used and more secondary functions. The SIMPL library API is very lean and clean.

In SIMPL vocabulary we speak of modules (Linux processes) as being one of two types:

- senders
- receivers

Senders compose and send structured messages to receivers. Receivers receive those messages and process them. Until the receiver replies to the sender, the sender is blocked. Until a receiver receives a message, it is blocked. This blocking provides a method of synchronization between processes, hence the S-Synchronous in SIMPL.

SIMPL senders are loosely related to *clients* in the client/server model. The SIMPL receiver is closest to a *server* in that model. In the client/server approach, the primary communication means is usually a TCP/IP socket. Local SIMPL receivers get their messages from a FIFO-synchronized shared memory area. This is not to say that a SIMPL message can’t be carried over a TCP/IP socket to a remote SIMPL receiver on another node. In fact, through the use of a surrogate process pair, SIMPL makes such a deployment completely transparent.

SIMPL, the library, treats messages as a raw collection of bytes. We are going to show that if that raw collection is tokenized we can greatly enhance the extensibility of SIMPL modules. This modular encapsulation simplifies things when a number of programmers work in parallel, thus shortening development time. SIMPL teams can also work very independently, often in different geographic locations. Isolating the algorithms into compact modules makes the software

easier to debug. Moreover, if a bug is introduced into the software, the affected area will be easier to isolate, retest, and reinstall. We are going to show that the SIMPL Testing Framework (STF) along with the modular SIMPL designs greatly facilitate testability.

## **1.2 Summary**

To summarize the Introduction we can say:

- SIMPL is a software design paradigm promoting encapsulation of complex programs into simpler and more manageable units.
- The SIMPL toolbox contains library functions which allow well-defined message passing between these encapsulated elements.
- SIMPL can be used locally and/or over a network of host machines.
- SIMPL modules are highly extensible and highly testable.
- SIMPL is a blocked message protocol.

## **1.3 Next Chapter**

In next chapter we will tell the story of SIMPL; how it came about, where it has been used and why it enjoys success.

# Chapter 2

## A SIMPL Story

### 2.1 The Story of SIMPL

The journey to SIMPL began in 1988 with a small advertisement from a Canadian company touting a multitasking, real-time OS for PCs of the day. That company was called QNX and the computers of the day were still mostly 286 and 386 machines. The advertisement said that if you called the toll free number they would ship you a self-booting floppy which showed QNX running the Tower of Hanoi game.

We (the authors) were working together at a small battery research company at the time. Our lab was automated with a second hand PDP-11 running Fortran and Assembly Language code. We were looking for a way to introduce PCs into the mix. QNX was successful beyond our wildest expectations. Not only did our 386 computer network (of two) manage the data acquisition and storage requirements for the two thousand battery test stations, it also served that data out to a network of Macintosh workstations. While doing all of this our little network also served as our code development platform.

At the apex we prided ourselves in being able to sit in a technical meeting of battery researchers in the morning and have a fully implemented and tested new software module operational by the end of the same day. What was the key to that ultrafast software design cycle? It was QNX's Send/Receive/Reply messaging scheme. Send/Receive/Reply allowed us to break software problems up into discrete and manageable modules. These modules could then be independently unit tested and finally deployed across our network. QNX was amazing technology considering we were talking about the late 1980s.<sup>1</sup>

---

<sup>1</sup>SIMPL takes the power of Send/Receive/Reply to the Linux operating system.

The SIMPL story resumes about ten years later. In the meantime, we had left the battery research company and started a software development company of our own. One of us (RF) became involved with an AIX project. This project used a middleware product (NetX) to provide an impressive messaging API. Despite the capabilities of this middleware we missed the simplicity of the QNX Send/Receive/Reply API. When the opportunity arose on the AIX project, a Send/Receive/Reply library was created using TCP/IP sockets for message transport. It was clunky and slow but it was only used to create test stubs. This effort was the seed which eventually grew into SIMPL.

A few years later, while between contracts on the AIX effort, we began thinking about a Linux implementation of Send/Receive/Reply. Why Linux? Having used QNX on many projects during that last ten years, we began to experience difficulties in dealing with the company from a business standpoint because QNX began to put increasing emphasis on large multiple license accounts. Furthermore, QNX licensing policies were not as open as some of our customers wanted. From a technical standpoint QNX was still the best OS we had ever used but getting timely device drivers for newer off-the-shelf hardware was increasingly a problem. Moreover, QNX was not always willing to provide the necessary compatibility information to write our own drivers. We needed to break free of this situation but didn't want to lose the QNX tools we came to know and rely on. Around the same time, Linux was beginning to hit its stride.

While the TCP/IP based implementation for AIX formed the starting point it was quickly rejected as being too complex. We decided to try to develop the Send/Receive/Reply messaging scheme from two very different directions, namely: a) SIMPL and b) SIPC. SIMPL is based on a user space FIFO synchronized shared memory scheme which remains at the core of the SIMPL library today. SIPC (Synchronous InterProcess Communications) was based on a kernel module approach. RF worked on the initial SIMPL and JC worked on the SIPC approach. It is worth mentioning that our friends at Cogent Real-Time Systems Inc. also made a functioning SIPC-type of messaging system and to our knowledge it is still operating today.

SIPC was initially very successful. It was perhaps half as fast as the QNX Send/Receive/Reply benchmark which we considered to be beyond expectations. Being largely kernel-based it was less subject to context switching slow downs resulting in decent performance. It was a treat to be able to write kernel modules without bureaucratic interference and the first version of the SIPC module was written for Linux version 2.0.36. As Linux itself progressed the issues of maintaining the SIPC kernel became obvious. With progressive major version releases the file operations structure was constantly changed. This meant that

the SIPC module had to be changed to mirror each new Linux version. This is not a big issue and judicious use of a make-file can handle much of this. But added to this is the fact that we had to support two separate libraries: the SIPC system call library and the user space program library as well, and this starts to become more complicated. Moreover, kernel-based programming is not the usual garden variety application programming. Other issues arguing against a kernel-based approach were questions of portability to other operating systems which support ANSI ‘C’ tools. Suffice to say that this was planned to be an open source project and we wanted future development to be unencumbered. Consequently, despite its faster messaging, we focused on the more general user-space approach of SIMPL. Given the local processing speeds of today’s CPUs one has to ask just how fast is fast anyway? As well, given that much of SIMPL’s applicability is network-based, network transmission speed slow downs more than overshadow local processing differences.

Around the time that the earliest SIMPL library was taking shape (1999), we were approached to do an embedded acquisition and control project. That project was originally specified for QNX but eventually landed on Linux. Thus the first user for the nascent SIMPL library was born. This early implementation of SIMPL proved to be surprisingly robust. This was probably due to the underlying simplicity of a user space FIFO plus shared memory implementation. Despite this simplistic approach SIMPL was benchmarking within an order of magnitude of QNX on the same hardware. It was during this early period (1999-2000) that SIMPL gained its first network surrogates. Those network surrogates made transparent TCP/IP transport of SIMPL messages possible.

The first of these surrogates is still known by its original name: tclSurrogate. It was thus named because its original intent was to allow Tcl/Tk applets to behave as well-formed SIMPL modules in a Linux network. The tclSurrogate and its accompanying protocol are still used to connect Tcl/Tk SIMPL applications across disparate OSs. Its uses are not restricted to that narrow niche. This protocol has been embedded in at least one deeply embedded network appliance (IO Anywhere). This same protocol has been used to allow a Visual Basic application to act as a SIMPL module.

The tclSurrogate was never a true surrogate in the QNX sense. To achieve true QNX style network transparency a new surrogate protocol and architecture was required. We also needed a simple mechanism to extend our SIMPL naming scheme to other network nodes. While QNX supported a network wide name propagation, we chose to extend the SIMPL name into a kind of SIMPL URL. Armed with this design the earliest implementation of the TCP/IP surrogates (2000) gave SIMPL its first claim to network transparency.

Everything worked and was quite robust. However, as SIMPL came to be deployed in more extensive applications it became clear that the original API wasn't as clean as it could have been. A memory mapped file implementation for the SIMPL sandbox was particularly stressed when SIMPL applications began to contain rapidly spawned processes.

As SIMPL 2.0 took shape (2001-2002), the API was cleaned up considerably and with some redesign the memory mapped file table was completely eliminated. This enhanced the robustness of SIMPL enough to merit the major version increment, but it also meant that SIMPL users faced their first major API change. SIMPL supported both APIs for a transition period. However, once developers recognized that the newer API was both simpler and superior, the old API was gradually deprecated and eventually dropped entirely.

SIMPL was extended to operate with MAC OS X. With this addition networks became more heterogeneous and the endian issue became relevant. Accordingly, the SIMPL network communications underlying remote message exchange which had always been binary in nature were then augmented to be either binary-based or character-based. The SIMPL library was originally only static but the ability to dynamically link was also added.

By 2003 a Python SIMPL library had been added. In this way, Python programs can send and receive messages to each other or to 'C' programs or anything else that is capable of SIMPL communications for that matter. Python has some very appealing programming properties; just ask any Python enthusiast!

By 2004 SIMPL had been deployed in numerous networked configurations. The original TCP/IP surrogate implementation was still reasonably robust but was beginning to show its warts. A redesign and simplification was in order. This TCP/IP surrogate simplification and redesign effort culminated in the SIMPL 3.0 release. A pleasant consequence of this simplification effort turned out to be a marked improvement in SIMPL 3.0's ability to clean up and recover from unplanned network failures. The core SIMPL API didn't change much for SIMPL 3.0 so the migration issues for existing users were minor.

As early SIMPL developers we almost always wrote modules in 'C', the language of the core library. Initially our focus as SIMPL developers was on mastering the art of problem decomposition into those SIMPL modules. This remains more an art than a science to this day. The SIMPL messages were almost always defined as 'C' structures. The layout of those messages, while an art in its own right, became second nature. It wasn't until SIMPL grew to embrace other languages (Tcl, Python, and Java) that the art of SIMPL message design became more formalized.

In this edition of the book we have captured one of our (JC) efforts to resur-

rect the kernel module for SIMPL, originally called SIPC. This new module has been renamed SRY.

As we write this, SIMPL is a relatively stable project. It has a Sourceforge presence which consistently ranks in the top 20% of its class. It has been downloaded by thousands of users. We don't always get informed where SIMPL is being used. We know it has been used for security systems, banking system test cradles, data acquisition in teaching labs, automated packaging lines, and ports of QNX code to Linux.

Contributions to the SIMPL project still occur from time to time. The most recent contributions have been in the direction of expanding SIMPL's reach to other platforms and programming languages. Such contributions have allowed SIMPL modules to be written in 'C', C++, Tcl, Python, and Java. One of us (RF) runs a no-fee online Linux programming tutorial.<sup>2</sup> Several thousand students have been exposed to the SIMPL way of programming through that course.

## 2.2 SIMPL Tales

In 1998 John Ousterhaut<sup>3</sup> wrote a paper on the evolution of languages used to create software programs. The paper discussed the differences between scripting languages (like Tcl) and system languages (like 'C' and C++). His thesis was that system languages were designed to efficiently handle complex data structures and algorithms while scripting languages were designed for gluing already working algorithms together. Scripting languages are typically interpreted rather than compiled, so they trade off raw execution speed for better time to market and code resiliency. In other words, scripting languages offer *good enough* performance and get a working application up and running more quickly than compiled languages do. In addition scripted applications are more readily changed as requirements for new feature sets come forward.

The SIMPL toolkit also excels at gluing. Modular SIMPL applications also represent a tradeoff of raw execution speed for better time to market and application resiliency when compared against monolithic application architectures.

SIMPL is not the appropriate tool to build a high performance finite element algorithm, but you could use SIMPL to build interfacing modules (such as a

---

<sup>2</sup>The iCanProgram online courses are available at:  
<http://www.icanprogram.com/nofeeCourses.html>.

<sup>3</sup>Scripting: Higher Level Programming for the 21st Century  
<http://home.pacbell.net/ouster/scripting.html>

GUI) to such an engine.

Low latency real time systems are not the domain of SIMPL, but SIMPL has been coupled with various Linux real time patches (eg. RTLinux) to form a moderate latency real time application. For an even higher performing real time system the Send/Receive/Reply architecture of SIMPL is modelled after one of the premier real time operating systems called QNX.

There are special tools for developing high transaction volume multithreaded applications, but for more moderate data acquisition applications SIMPL is ideally suited. SIMPL has been used to build a testing cradle for one such high transaction banking system. In that instance the flexibility and deterministic nature of the Send/Receive/Reply allowed that cradle to be built and deployed in a matter of weeks. SIMPL has been used in many low transaction rate (less than 100 samples per second) data acquisition applications. In one such application SIMPL was used to create a polling hardware handler, datastore, replicator, configurator, and applet based GUI using ‘C’, Tcl/Tk, and RTLinux from spec to deployment in less than six weeks.

Deeply embedded appliances typically require highly efficient code. Linux and SIMPL are too high level for these applications. However, SIMPL has been used to glue one such appliance to a broader SIMPL application. As proof of SIMPL’s resilience a demo of a SIMPL enabled embedded appliance was assembled offsite in one week, unit tested without access to the larger SIMPL application to which it was to interface and deployed against that larger application without any changes in less than one hour.

SIMPL doesn’t often get used to re-architect or port many legacy applications, but SIMPL has been used to extend the reach of such applications. In one instance SIMPL was used to join a laboratory data acquisition system to a legacy UNIX based graphing package. In fact SIMPL has even been used to extend the reach of legacy QNX applications which would otherwise be too expensive to port.

You could use SIMPL for single programmer projects, but the SIMPL toolkit really excels in complex projects which require a multidisciplinary/multilanguage team of developers. In one such project, a team of four programmers was able to build a SIMPL application consisting of a touch screen man/machine interface, PLC interface, printer managers, mainframe XML host interfaces and state machine logic engine in four weeks from beginning to factory floor deployment. The ability to modularize the application and then develop and test each SIMPL module in isolation was key to the success that project. The fact that different modules could be created in different languages (Tcl/Tk, ‘C’, Python/Tk, and Java/Swing) was also key to the time to market.

You would not use SIMPL to build a dynamic website, but SIMPL has been used to build several Internet distributed applications (eg.building/home automation). Many other SIMPL applications have been distributed across multiple nodes in a local TCP/IP network. Through the use of surrogates the SIMPL developers need not concern themselves with any of the details associated with transporting messages across networks. SIMPL modules are developed locally and then deployed network wide, often without even a recompile.

Web centric designs employing web services architectures are increasingly popular. SIMPL doesn't substitute for web services toolkits, but it has increasingly been used as a **web services lite** in systems where resource constraints can't allow full web services overhead. SIMPL is ideally suited to creating interfaces between existing web services applications and embedded devices. This is especially true now that Java-SIMPL has reached a sufficient level of maturity.

Most SIMPL applications are built under Linux. Linux is the OS that SIMPL was designed for. However, the SIMPL library has been ported to other UNIX-like operating systems (eg. AIX and MAC OS X) and reports are that it works well there. Many SIMPL applications are hybrids between multiple operating systems. (eg. Windows and Linux). While in those cases the Windows system is *tricked* into thinking that it knows how to run SIMPL, neither the developer nor the user need to concern themselves with those details. The illusion is seamless. The SIMPL developer can now choose between Java, Python, or Tcl when creating code to run on hybrid networks which include Windows.

Not all SIMPL projects succeed. SIMPL promotes a problem decomposition strategy which, if done well, will amplify the chances of success. However, if done poorly it can achieve the opposite. Furthermore, not all project management structures are in tune with resilient architectures that can be added to and changed with ease. Many front loaded, top down design paradigms assume that all has to be known about a problem before coding can begin. Change control is often a cumbersome and bureaucratic process designed in part to *discourage* changes to the application. These structures are poorly adapted to take advantage of the power of SIMPL. It is no fluke that SIMPL is an open source project. The open source design paradigm is ideally suited to the SIMPL way.

## 2.3 About Programming Paradigms

There is quite a bit of information available regarding so-called *Programming Paradigms*. It appears to be a somewhat opinionated subject. This causes a

certain amount of disparity and variation between what is considered to be a programming paradigm and otherwise. Our interest here is to take a look at where SIMPL might fit with respect to several generally accepted programming paradigms.

Let's begin by stating what we consider the phrase *Programming Paradigm* to mean. Simply put, it can be thought of as a programming viewpoint. It is the point of view that a programmer takes when planning and writing code that performs some function. Different programming languages are more or less amenable to these viewpoints. If a certain viewpoint or paradigm is mandated, then it is likely that certain programming tools and languages will fit the situation better than others. In reverse, if the programming language for a given software project has been decided, then the appropriate paradigm(s) will have to be used. It's a two-way street between programming paradigms and the programming languages that support them. If you ask veteran programmers which languages they prefer and why, they will name some languages that they find more adept for following a certain *programming style*. This programming style is usually some form or variant of one or more programming paradigms. Paradigms are not necessarily mutually exclusive. Some programming languages are suitable for more than one approach, while others are less so.

The following is a fairly short list of generally accepted programming paradigms. According to some it will be incomplete while according to others the list will be too long.

1. **Imperative** - describes a temporally ordered sequence of program statements. The earlier statements being executed in time before the latter statements. A variant of this is called **Procedural Programming** when groups of statements are collected together in the form of separate procedures.
2. **Object Oriented** - describes an arrangement where data and the methods for manipulating the data are kept together in a quantity called an object.
3. **Logical** - instead of telling a program how to calculate some value of interest, the value is described by a number of declarations regarding its qualities and the program produces such a value by logical inference. This can be seen in SQL calls as carried out by a database engine for example. **Declarative** can be viewed as a subset paradigm of logical.
4. **Functional** - describes a system where the fundamental building block of a program is a function. That said, let's now define what we mean by a

function. First of all, it's not a function as might be defined in 'C'. That would be a better fit in *Procedural Programming*. Function in this context is more like a mathematical formula which defines a relationship between two or more items.

So, where does SIMPL fit within these paradigms? Certainly SIMPL can be set within the *Imperative*. That this is so can be seen from the fact that Send-Receive-Reply is very deterministic and temporally ordered. There is also a SIMPL library for C++ which can easily embrace the *Object Oriented* approach. The value of SIMPL within a group of objects is often a matter of opinion. As for the rest, let your imagination take flight.

## 2.4 Summary

The story of SIMPL can be summarized as follows:

- SIMPL takes its form based on the message passing paradigm used by the QNX operating system in the 1980s.
- SIMPL takes this operational structure from the QNX platform to other platforms, most notably Linux.
- SIMPL uses a surrogate approach to broaden its scope from message passing communications between processes local to one host to message passing between processes on different host machines.
- SIMPL libraries exist so far for 'C', Python, Java, and Tcl.
- SIMPL surrogates exist for the TCP/IP and RS-232 protocols.

## 2.5 Next Chapter

In the next chapter we are going to discuss the concept of softwareICs and how they are to software designs what traditional electronic ICs are to PC board layouts.



# **Part II**

# **SIMPL Elements**



# Chapter 3

## SoftwareICs: Theory

”Complexity must be grown from simple systems that already work.” –Kevin Kelly <sup>1</sup>

### 3.1 SoftwareICs

Computer hardware is complex, so is computer software. In fact as the cost per unit of computational horsepower continues to decrease, user demands on software applications have become ever more complex. Unfortunately, with all this increased software complexity comes problems associated with creating robust software in a cost effective and timely manner.

A number of years ago hardware engineers faced similar issues. Before the advent of Integrated Circuits (ICs), electronic circuits were built up of discrete components (transistors, resistors, capacitors etc.) all interconnected with printed circuit boards and wires. As complexity of electronic circuits increased it became increasingly difficult for hardware engineers to manage this intricacy. This difficulty adversely affected the scope and cost of products as well as their time to market.

With the advent of ICs much of this complexity began to be hidden inside the chips themselves. Thousands of discrete components were replaced by a silicon chip with well-defined functionality and pinouts. The inner workings of the chips themselves were incredibly complex, but the integration of the chip

---

<sup>1</sup>Out of Control: The New Biology of Machines, Social Systems, and the Economic World by Kevin Kelly, Addison-Wesley Pub., 1994, ISBN 0-210-48340-8

into any given design was much more manageable. The job of designing complex functionality into products became much easier.

Complexity was encapsulated and miniaturized. This complexity encapsulating innovation is what has allowed hardware engineers to make such spectacular progress. The concept of a hardware design as an interconnected series of islands of complexity represented a huge architectural shift. The advent of ICs allowed the hardware designer to use a given chip in a product design without having to understand circuits within the chip itself. As long as the designer conformed to the specifications of the external interface to the chip (pins), the chip would react in a very predictable manner.

This is what encapsulation of complexity offers:

- Complexity hiding.
- Predictable/reproducible behavior.

Software is still created out of discrete components and painstakingly hand-coded to form an application. Software has hit a scope and complexity wall similar to that faced by the pre-IC hardware world. Custom software projects often go over budget and underdeliver on ever more complex user requirements.

Software engineers should look to their hardware counterparts for better approaches to managing complexity. What software needs is that equivalent encapsulating technology to the IC: what we are going to call the *softwareIC*.

Object Oriented Programming (OOP) languages such as C++ or Java attempt to attack the software complexity problem at the programming language level. For some classes of problems such as GUI and database design they have been successful. For whole other classes of problems OOP languages have only succeeded in shifting complexity to other areas in the software development chain such as testing, toolsets, class library design, maintenance or the learning curve. At best objects in software design offer partial complexity hiding. The software designer still has to know and master a complex language (e.g., C++) in order to be able to wire objects together into a product. Often the toolkit itself dictates which language is used for all aspects of a particular application.

In our opinion, true encapsulation of complexity behind a universally simple and extendable API is required to produce a softwareIC. The software designer should not have to master a complex object-oriented language and toolset in order to be able to *wire together* these softwareICs. At the very least the software designer should be able to choose the wiring language independent of the chip language.

With the advent of wide area networks such as the Internet, software applications are increasingly distributed and multiprocessed. In such situations the choice of a message passing paradigm often unintentionally introduces a degree of unpredictability and randomness into a software product. It has been said that client/server software design does not handle complexity well. If the client and the server are not regularly brought into synchronization it is exceedingly difficult to replicate or predict all possible states of a multiprocessing system. Much depends on the environment and timing of these processes. Modules may work fine in one environment and suffer sporadic failures in another. This leads to increased code complexity as well as increased testing and maintenance costs, all resulting in a poorer quality software product. Often the blame is incorrectly directed at the multiprocess design paradigm. We agree with the original designers of Send/Receive/Reply synchronized messaging who believed that the answer lies in forcing a state-machine-like synchronization to occur on each message pass.

Many have argued that this blocking/forced synchronization introduces unnecessary complexity into a message exchange, but when properly applied it achieves exactly the opposite effect. By forcing synchronization to occur at each message pass, one finds that multiprocess applications behave in a very predictable and reproducible manner. Gone are the timing and environmental effects that plague nonsynchronized message passing schemes. After all, one of the oldest blocking/forced synchronized message passing schemes is the simple function call using the stack for message transport. When dealing with complex applications, predictability and reproducibility of behavior represent a great strategic advantage.

With Send/Receive/Reply systems, the sender is blocked during message transmission and explicitly unblocked by the receiver process with the reply. As such it is very easy to arrange to transport these messages over a variety of media (including some which are *slow*). The messages could be exchanged via shared memory if the processes were on the same processor, or via the Internet if the processes were physically separated or on a serial line in a dial up or radio modem situation. While the throughput of the collective of processes would be affected by the message transmission speed, the performance would be predictable and reproducible.

Software *must* be predictable for reliable testing. Nothing makes software QA people wish for a career change more than an application which exhibits unpredictable and unreproducible behavior. Predictability and reproducibility of behavior is an essential requirement for a softwareIC.

Fortunately, modern multitasking operating systems provide us with a ready

analogue to the hardware integrated circuit: a user space process running in a protected mode. If one of those processes encounters a fatal error it rarely brings down the whole machine even if something sends a process into a locked state.

Many real time operating systems (RTOSs) have pioneered the use of user-space processes as an encapsulation scheme. One of the oldest to use this scheme is QNX. Since 1980, QNX has released a continuous series of innovative operating systems that were based upon a set of cooperating processes using a Send/Receive/Reply messaging paradigm. QNX's approach to kernel design differs greatly from that used in Linux and we do not wish to reinflame the infamous microkernel vs. monolithic kernel debate. Suffice to say that we believe that the process model and the Send/Receive/Reply messaging paradigm first pioneered by QNX offers the key ingredients of a successful softwareIC.

Fortunately in the SIMPL toolkit, with its QNX heritage, we have an ideal candidate for the creation of a softwareIC on Linux systems. The Linux user space process is our chip, the SIMPL Send/Receive/Reply messaging is our pinout and the message bytes themselves are the interconnecting wires.

## 3.2 SIMPL Axioms

When development of the SIMPL library and utilities began, the concept of a softwareIC had yet to be articulated. Nonetheless from its inception, the development of the SIMPL toolkit and its subsequent applicability to the softwareIC concept has been guided by a set of principles. These principles influenced and arbitrated all our major design decisions with SIMPL.<sup>2</sup>

The SIMPL axioms are:

**Anonymity** Neither a SIMPL sender (initiator of messages) nor a SIMPL receiver (consumer of messages) can readily discover the name of their SIMPL communications partner. This anonymity forms the basis of the ready testability of SIMPL modules using stubs as partners.

**Need to know** A SIMPL sender doesn't need to know where its communication partner is (locally or on a network). A SIMPL sender shouldn't need to know the details of a communications partner's algorithm. It should only need to know the message API. Response timing is the responsibility of the SIMPL receiver and not the SIMPL sender. For example, a SIMPL sender

---

<sup>2</sup>Webster's Dictionary defines axiom as: *an established rule or principle or a self-evident truth.*

should not need to know if a SIMPL receiver is going to hold back a reply or not (a concept known as reply blocking). A SIMPL communicating pair would have to agree on such things as byte ordering and message structures, but to the SIMPL transport layer these are just a collection of bytes. Furthermore, the SIMPL layer should never have to peek into this message structure for any reason.

**Democracy** SIMPL message exchange sequencing between primitive pairings is always on a first in first out basis. However, with appropriately constructed SIMPL softwareICs you can achieve priority store and forward messaging algorithms. That feature is just not part of the basic SIMPL message transport.

**Compile Once, Deploy Anywhere** A SIMPL module should not need to code anything with respect to the message transport. A SIMPL module should be able to be compiled, tested locally and then deployed across the network without any recompile.

**KISS** Whenever we were faced with decisions with multiple paths we always endeavoured to choose the simplest path which had any useful value and implement that. The SIMPL API is deliberately basic. The choice to use a colon delimited composite SIMPL name to allow network transparency is also deliberately basic.

**Timeouts** Over the years many users have asked for SIMPL *Send* timeouts. We've come to believe that no application should need this feature. Whenever we've arrived at a design where we thought we needed such a timeout there was always a better way of looking at the layout which negated that need. There are timeouts internal to SIMPL surrogates which endeavour to add robustness to the message transport algorithm. When these timeouts kick in they always appear to the SIMPL modules as *Send* or *Reply* failures.

**Security** We believe that SIMPL applications themselves should look after any security aspects required. In other words, if encrypted messages and certificates are required they belong at the SIMPL application layer and not as part of SIMPL itself.

### 3.3 SIMPL as a SoftwareIC Toolkit

The softwareIC as a concept is more general than SIMPL. SoftwareICs could be written with toolkits other than SIMPL. We have found that SIMPL is a great toolkit for creating general purpose softwareICs on a Linux platform. Here are some of the reasons why.

The SIMPL library, and many of the projects which use SIMPL have been written in ‘C’. However, hooks to allow the creation of SIMPL modules in Python, Java, Tcl, and C++ have been made available as the SIMPL toolkit has matured. This means that a SIMPL softwareIC can be written in the most appropriate language for its algorithm rather than the language dictated by the choice of an OOP toolkit. SIMPL modules created in these different languages can be mixed in a given SIMPL application. The language used to write the SIMPL softwareIC itself in no way dictates the language of another SIMPL module with which interaction takes place. Furthermore, a given SIMPL process has no way of discovering which language was used to construct another softwareIC with which it is exchanging a message.

A properly constructed SIMPL softwareIC has no need to know or discover the physical location of its exchange partners. This means that the same binary image can be tested with local message exchanges and then deployed with remote message exchanges (see chapter 7). Overall collective application performance would differ but the individual softwareIC would not need to change in any way. In may instances even a recompile is unnecessary.

A SIMPL softwareIC is completely insulated from the internal algorithm of an exchange partner. This means that test stubs can be created which completely simulate and replicate the environment surrounding a softwareIC. In particular, error conditions which would be costly or difficult to reproduce in the full system can be simulated in a test environment. This means that SIMPL softwareICs can be rigorously tested before being deployed (or redeployed) in the real world application.

SIMPL softwareICs lend themselves well to projects with multiple developers. SIMPL applications of any complexity contain multiple SIMPL modules. Once the messaging API is agreed upon, work can proceed in parallel with each SIMPL module in a given application. The implementation details of a SIMPL enabled process cannot affect any interacting process provided that implementation conforms to the agreed upon message API. While a poor softwareIC implementation will affect the overall application performance adversely, the application will still operate. Once a poor algorithm has been identified it can optimized in isolation from the rest of the application. Once fully tested, this reworked softwareIC can

be inserted into the application without even recompiling the adjacent software-ICs.

When faced with designing an application using the SIMPL toolkit, it used to be that one would sketch out a *circuit board* with the basic SIMPL primitives: senders and receivers. This works for basic projects, but over the years the SIMPL project has accumulated an extensive codebase of useful frameworks and example code which we have called our softwareICs repository. Now when SIMPL designers begin designing complex applications, they can sketch out *circuit boards* with senders, receivers, proxies, broadcasters, agencies, relays etc. SIMPL softwareICs have one thing which distinguishes themselves from their hardware cousins: they are inherently extendable and adaptable.

## 3.4 Summary

In summary here is why SIMPL softwareICs represent a better way to design complex software applications:

- In principle SIMPL softwareICs can be written in any language.
- The language used to write the softwareIC itself in no way dictates the language of another softwareIC with which interaction takes place.
- A properly constructed SIMPL softwareIC has no need to know or discover the physical location of its exchange partner.
- A SIMPL softwareIC is completely insulated from the internal algorithm of an exchange partner.
- SIMPL softwareICs lend themselves well to complex projects with multiple developers.
- The public SIMPL softwareICs repository represents an increasingly useful body of seed code for any given project.

As was the case with the advent of hardware ICs, this softwareIC approach to software development lowers project risk/cost and allows ever more complex applications to be brought to market in a timely manner.

## 3.5 Next Chapter

In the next chapter we will study the details of the core library functions of SIMPL. We are going to look at how local SIMPL communications are added into program code and how processes pass messages.

# Chapter 4

## SIMPL Core

### 4.1 The Core Elements of SIMPL

Previously we introduced you to the SIMPL design paradigm. We also gave you a flavour for the motivation in creating SIMPL with a short description of its history. In this chapter we are going to dive a little deeper into the core SIMPL library.

Before beginning that discourse, allow us to use an analogy. Two co-operating SIMPL processes are like two individuals about to engage in a phone conversation. The conversation initiator (SIMPL sender) would need to know the phone number (receiver ID) of the other party (SIMPL receiver). Normally this conversation initiator would know some identifying information about the intended conversation partner: likely the person's name (SIMPL name). To obtain the phone number (receiver ID) a look up by name (*name\_locate*) in either an address book (FIFO\_PATH) or a phone book would be required. To be able to cross-reference the person's name with a phone number meant that that person would have previously supplied information to the indexing organization (*name\_attach*). Armed with the phone number, the phone conversation can be initiated by dialing (*Send*). When the intended conversation partner picks up the call (*Receive*) a conversation can ensue. At this point our analogy breaks down somewhat because a SIMPL conversation is more like a two way radio conversation than a phone conversation. In a two way radio call only one person can speak at a time and the conversation takes on a talk-response-talk-response (Send - Reply - Send - Reply) characteristic.

In the following section we will be referring back to this telephone analogy as we discuss SIMPL communications restricted to processes running on one host

computer.

## 4.2 SIMPL Programming Languages

A SIMPL application consists of two or more interacting SIMPL modules. At the heart of each SIMPL module is the SIMPL core library. The core library for SIMPL is written in ‘C’ and designed to run on a Linux OS. However SIMPL modules can be written in a number of supported languages (‘C’, C++, Python, Java, and Tcl). To enable this, shared libraries which wrap the core ‘C’ library calls are maintained for each of the non-‘C’ languages. In other words a Java-SIMPL module running on a Linux OS is ultimately still calling the core SIMPL ‘C’ library.

SIMPL modules themselves come in two flavours: the Linux OS and the non-Linux OS version. In the Linux OS flavour all the code is contained within one executable which (statically or dynamically) links the SIMPL core library. The non-Linux flavour implements the SIMPL module as two interacting parts: a generic Linux part which links the SIMPL core library and business logic part which connects using a TCP/IP socket protocol (see Appendix J). Each of the non-‘C’ languages for SIMPL sport at least two variations of shared library. One wraps the core ‘C’ SIMPL calls as above; the other implements the tclSurrogate TCP/IP protocol to allow the business logic half of a SIMPL module to run seamlessly on a non-Linux OS.

Since the core SIMPL library is designed for Linux, most SIMPL applications are targetted for that platform. This is most certainly true for SIMPL applications which are deployed on a single node. The most popular language for creating a Linux SIMPL module is still ‘C’. In heterogeneous networked SIMPL applications the non-‘C’ languages are popular for the business logic portions of non-Linux SIMPL modules. Enabling SIMPL modules to be deployed on a Windows node is by far the most common use for the non-‘C’ SIMPL supported languages.

## 4.3 Local SIMPL

The purpose of this section is to discuss the inner workings of SIMPL on a *local* level. By *local* we are referring to the method of operation used by SIMPL to enable communications between processes running on a single host computer. Core SIMPL is only functional at a local level. In order to extend this core

functionality across a network, we rely on the notion of surrogates. We will be discussing surrogates at length later on in the book.<sup>1</sup>

### 4.3.1 SIMPL Names

*ALL* processes that use SIMPL *must* be uniquely named per host computer. In our telephone analogy this is equivalent to saying that there can only be one *John\_Smith* listed for any given city. The SIMPL library will enforce this rule, so it is up to the SIMPL programmer to ensure a set of unique SIMPL names for the modules running on any given node. For what follows let's call this unique name the "SIMPL name". It is a string of characters and is the analogue to the name of the person we are trying to call.<sup>2</sup> A SIMPL process names itself by making the **name\_attach**("SIMPL name") library call. This is normally the first order of business in any SIMPL program. Once a SIMPL module is named it can engage in SIMPL communications.

To initiate this communication a SIMPL sender process will make the **name\_locate**("SIMPL name") function call to return a unique integer called the SIMPL ID. In our telephone analogy the SIMPL ID is the telephone number and the **name\_locate** call is the act of looking the number up. Once a sender process has obtained the unique SIMPL ID of a receiver process, it is then able to initiate a message exchange via the **Send** library call.

The above can be summed up in the table below:

Telephone	SIMPL
telephone client name	SIMPL name
telephone listing	name attach
telephone number	SIMPL ID
telephone number lookup	name locate

Table 4.1: Telephone/SIMPL Analogy

### 4.3.2 SIMPL Messaging

In a local SIMPL message exchange, the message data is written into an area of shared memory owned by the sender. This is true for both the outgoing message

---

<sup>1</sup>For detailed information on surrogate processes see Chapter 7.

<sup>2</sup>The size of the root SIMPL name is dictated by the MAX\_PROGRAM\_NAME\_LEN define simplDefs.h which currently defaults to 31 characters.

written by the sender and the response written by the receiver.<sup>3</sup> This memory is *shared* because both the initiator and recipient will be interacting with it. Since two parties are writing to and reading from the same shared memory area, some mechanism must be employed to ensure uncontested access. SIMPL uses a mutually exclusive process blocking scheme to regulate access to the shared message memory. See Figure 4.1.<sup>4</sup> This shared memory is identified on the host system by a unique integer called the shared memory ID. The handoff of access to the message data is coordinated by the exchange of this shared memory ID over a FIFO owned by the message recipient. The shared ID handoff and subsequent access to message data by the recipient constitutes the local SIMPL message *transport*.

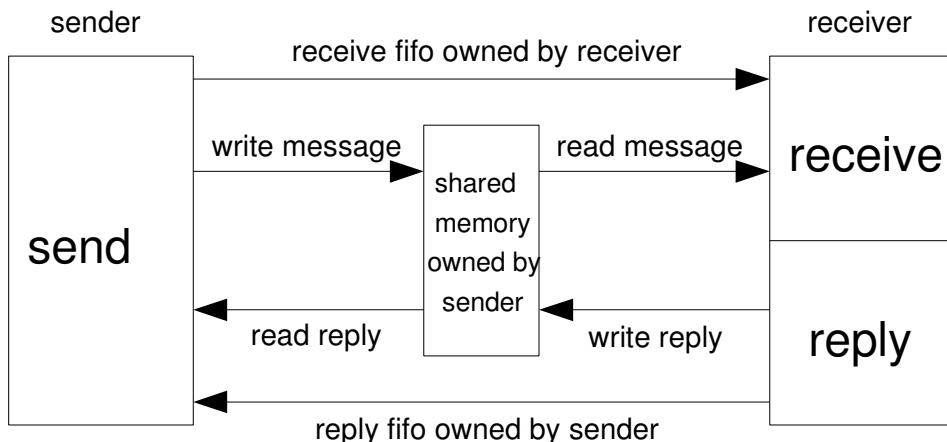


Figure 4.1: Send/Receive/Reply

The receiver is receive-blocked while it awaits the arrival of a shared memory ID on its FIFO. In effect, this receiver process does nothing until it reads a shared memory ID written to the other end of the FIFO by a sender. Likewise the sender will immediately reply-block on its FIFO after the shared memory ID has been written to the receiver's FIFO. This mutually exclusive blocking on a pair of FIFOs is what achieves full synchronization of message and response in a SIMPL transaction.<sup>5</sup> When a shared memory ID arrives at the receiver FIFO,

<sup>3</sup>The **Send** call actually allocates the shared memory based on the larger of the two supplied size parameters.

<sup>4</sup>It is important to note that this process blocking scheme is also what enables SIMPL messages to be transported seamlessly over network links by surrogate processes (see Chapter 7).

<sup>5</sup>Blocking in this case means that the process will cease to run until it is explicitly unblocked

the receiver process is unblocked, copies the message from the sender's shared memory, processes the message in some pre-ordained way and then replies back to the sender. The receiver can utilize the same shared memory area owned by the sender to post its response with full confidence that the sender will not be modifying that area in its reply-blocked state. The sender will remain reply-blocked until the receiver has completed the transaction and writes the shared memory ID to the sender's reply FIFO. At this point the receiver goes back to the receive-blocked state trying to read its FIFO. Since the receiver's FIFO is a natural queuing device, we have natural serialization of the message flow into a SIMPL receiver from multiple senders. Since the message data is transported in shared memory owned by the sender, this serialization also achieves a natural queuing of SIMPL messages.

In summary, in a SIMPL transaction receive-blocked receivers get unblocked by a ***Send*** from a sender and reply-blocked senders get unblocked by the ***Reply*** from a receiver. In SIMPL terminology we call this mechanism Send/Receive/Reply or SRR for short.

Keeping Figure 4.1 in mind, let's look at what happens during a typical message exchange.

1. The receiver is receive-blocked; it awaits an integer message via its receive FIFO.
2. The sender loads its shared memory with the contents of a data message.
3. The sender writes its shared memory ID to the FIFO via the integer message.
4. The sender now becomes reply-blocked and awaits a response from the receiver on the reply FIFO.
5. The receiver reads the integer message from the FIFO. The value of the integer tells the receiver what shared memory to open.
6. The receiver opens the shared memory, reads the message header and the appropriate contents.
7. The receiver processes the message and forms a reply for the sender.
8. The receiver replies back to the sender by loading the sender's shared memory with the reply data message and then writes an integer message on the reply FIFO.

---

by the initiator of the message.

9. The sender reads the integer message on the reply FIFO, becomes unblocked and then reads the reply data message from the shared memory and processes it.

There are three important items to emphasize. Firstly, the shared memory is owned by the sender, not the receiver. The main reason for this is that if it were owned by the receiver, and there was more than one sender, then messages could be overwritten. This shared memory is allocated when the sender sends its first message and is deallocated if the sender terminates. Secondly, the reading and writing of the FIFOs is atomic, as evidenced by the integer message referred to above. Making the FIFO *message* atomic ensures that any messages written to and read from the FIFOs are performed with a minimum of overhead. This is the mechanism that SIMPL uses to synchronize the sending and receiving of data messages. Thirdly, the receive FIFO is owned by the receiver but the reply FIFO is owned by the sender. In this way, messages to a receiver get naturally queued but any given sender only receives one reply FIFO message at a time.

In order for a program to use SIMPL, the shared memory and FIFOs must already be configured within the program. This configuring is the task of the ***name\_attach*** call. All programs which desire to utilize SIMPL must make this call prior to any other SIMPL function calls such as ***Send***, ***Receive***, ***Reply***. The purpose of this call is to set up the necessary elements for all local SIMPL communications. The important part of this functionality is that the two FIFOs are made in the FIFO\_PATH directory (see Chapter 10): the first is called the receive FIFO and the second is called the reply FIFO. The actual names of the FIFOs are a combination of the SIMPL name and the PID (process identification) of the calling program; hence the FIFO names are themselves unique to the computer. This uniqueness is of paramount importance as this provides the SIMPL *addressing* scheme for getting messages to and from the correct processes.

If you recall in our analogy above, the ***name\_attach*** call is analogous to registering your name with the phone company so that it will appear in the phone book. The analogue to looking up the phone number in that phone book is the ***name\_locate*** call. You couldn't look someone up in the phone book until that person had registered their information with the phone book publisher. Therefore in a SIMPL system it is imperative that the relevant processes are started in the correct order. Since a SIMPL sender must perform a ***name\_locate*** call to open the prospective receiver's receive FIFO for writing, that SIMPL receiver must have already performed a ***name\_attach*** to create this FIFO. The ***name\_locate*** call returns a file descriptor to the receiver's receive FIFO.

At this point we should recap. Let's do this by looking at how a receiver and

a sender would set up and use SIMPL.

### Receiver Steps

- R1.** Allocate adequate memory for incoming and outgoing messages.<sup>6</sup>
- R2.** The receiver performs a ***name\_attach*** call. This enables all of the necessary mechanisms required to use SIMPL on the receiver's side.
- R3.** The receiver makes a ***Receive*** call and becomes receive-blocked, awaiting any incoming messages from senders.
- R4.** Upon receiving a message, the receiver presumably processes the message and forms a reply message if required.
- R5.** The receiver makes a ***Reply*** call and returns the reply message to the sender. Note that sometimes a reply message is not required by the sending process; in this case a NULL response (a message of zero size and no content) is sent. The Reply must be made in order to unblock the sender.
- R6.** When the program is finished, the ***name\_detach*** call releases all SIMPL components. This call is present for completeness and is a good idea to include in the program but it should be understood that under a normal program termination any SIMPL components will be released by default.

### Sender Steps

- S1.** Adequate process memory for outgoing and incoming messages is allocated.
- S2.** The sender performs a ***name\_attach*** call. This enables all of the necessary mechanisms required to use SIMPL on the sender's side.
- S3.** The sender performs a ***name\_locate*** call to open a SIMPL connection to the intended receiver.
- S4.** The sender composes the message to be sent.
- S5.** The sender makes a ***Send*** call to the receiver and becomes reply-blocked awaiting the reply message from the receiver.
- S6.** Upon obtaining a reply, the sender carries on with its programming, presumably using the contents of the replied message.
- S7.** See R6 above.

---

<sup>6</sup>adequate memory is at least equal to the largest expected message size

## 4.4 SRR Example

In the following example we are going to illustrate how a sender program and a receiver program use SIMPL in order to communicate with one another. The sender will send ten messages to the receiver consisting of an integer valued from 1 to 10, one at a time. The receiver will square the integer and reply the result to the sender who will print out the original number and the squared number. This example is not meant to be a realistic SIMPL problem but merely to demonstrate how SIMPL communications work from a code standpoint.

Note that both the receiver and the sender programs are written in ‘C’, Tcl, Python, and Java; they are arranged in no particular order. This is because *any* sender can send the message to *any* receiver. The Tcl sender program can send the message to the ‘C’ receiver program if desired or the Python sender can send the message to the Tcl receiver or the ‘C’ sender can send the message to the ‘C’ receiver and so on. Feel free to choose the sender/receiver code snips written in the language you are most comfortable with because they all do the same things and carry the same commentary.<sup>7</sup>

In each code snip the *Receiver Steps* R1-R5 and the *Sender Steps* S1-S5 listed earlier are indicated as comments on the appropriate lines when applicable.

---

<sup>7</sup>If you want to follow along without transcribing these examples the source is available online at <http://www.icanprogram.com/simplBook>.

### 4.4.1 The Receivers

#### The Tcl Receiver

```

1  # Tcl receiver: program called tcl_receiver.tcl
2
3  #!/usr/bin/tclsh
4
5  # initialize variables
6  set myName RECEIVER
7  lappend auto_path $env(SIMPLHOME)/lib
8  package require Fctclx
9
10 # perform simpl name attach
11 set myslot [name_attach $myName] ;# R2
12 if { [string compare $myslot "NULL"] == 0 } {
13     puts stdout [format "%s: cannot attach name" $myName]
14     exit
15 }
16
17 while { 1 } {
18     # receive incoming messages
19     set buf [Receive] ;# R3
20     binary scan $buf i1i1 fromWhom nbytes
21     if {$nbytes == -1} {
22         puts stdout [format "%s: receive error" $myName]
23         continue
24     }
25     binary scan $buf x8i1 inNumber
26
27     # calculate square of sent number
28     set outNumber [expr $inNumber * $inNumber] ;# R4
29
30     # reply squared number to sender
31     set rMsg [binary format "i1" $outNumber]
32     set rBytes [string length $rMsg]
33     Reply $fromWhom $rMsg $rBytes ;# R5
34 }
35
36 name_detach ;# R6
37 exit

```

line 3 Invoke the Tcl interpreter.

line 6 Set unique SIMPL name.

lines 7-8 Include SIMPL library package.

lines 11-15 The program performs a ***name\_attach***. Recall that this is required prior to any other SIMPL library calls. The argument in the ***name\_attach*** call is the SIMPL name chosen to be unique on the local host. Again, recall the phone book analogy; this is like adding the receiver's name to the phone book.

lines 17-34 This is an infinite loop allowing messages to be received, processed and replied to continuously. Many receiver type programs have this format.

line 19 This is where the program receives incoming messages and places them one at a time into an input buffer.

line 20 The SIMPL ID of the sender and the size of the message are extracted from the input buffer. The receiver needs to know who to reply back to and how many bytes to read from the buffer to obtain the message.

lines 21-24 The incoming message is checked for problems.

line 25 The incoming message content, an integer, is extracted from the input buffer.

line 26 The integer is squared.

lines 31-32 The outgoing message is composed.

line 33 The outgoing message is replied back to the sender.

line 36 The SIMPL program components are removed.

## The Python Receiver

```
1 # Python receiver: program called py-receiver.py
2
3 #! /usr/bin/python
4
5 # import necessary modules
6 import sys
7 from csimpl import *
8
9 # initialize necessary variables
10 me = "RECEIVER"
11
12 # perform simpl name attach
```

```

13 retVal = nameAttach(me, 1024)                      # R1/R2
14 if retVal == -1:
15     print("%s: name attach error-%s" %(me, whatsMyError()))
16     sys.exit(-1)
17
18 while 1:
19     # receive incoming messages
20     messageSize, senderId = receive()                 # R3
21     if messageSize == -1:
22         print("%s: receive error-%s" %(me, whatsMyError()))
23         sys.exit(-1)
24
25     # unpack the number from the incoming message
26     inNumber = unpackInt(SIMPL_BIN)                   # R4
27
28     # calculate square of the number
29     outNumber = inNumber * inNumber                  # R4
30
31     # pack the square of the number into the reply message
32     packInt(outNumber, SIMPL_BIN)                    # R4
33
34     # reply squared number to sender
35     if reply(senderId) == -1:                         # R5
36         print("%s: reply error-%s" %(me, whatsMyError()))
37
38 nameDetach()                                         # R6

```

line 3 Invoke the Python interpreter.

lines 6-8 Import necessary Python and SIMPL modules.

line 11 Set unique SIMPL name.

lines 14-17 The program performs a ***nameAttach***. Recall that this is required prior to any other SIMPL library calls. The argument in the ***nameAttach()*** call is the SIMPL name chosen to be unique on the local host. Again, recall the phone book analogy; this is like adding the receiver's name to the phone book. The 1024 is the maximum size that a received or replied message may be in bytes.

lines 19-40 This is an infinite loop allowing messages to be received, processed and replied to continuously. Many receiver type programs have this format.

line 21 This is where the program receives incoming messages. The first returned value is the size of the message and the second is the sender's SIMPL ID.

lines 22-24 The incoming message is checked for problems.

line 27 The sent number is unpacked from the incoming message.

line 30 The number is squared.

line 33 The squared number is packed into the outgoing message.

lines 36-37 The reply call replies the result of the squaring process to the sender.

The argument represents the sender's SIMPL identification which was gotten from the earlier ***receive*** call.

line 39 The SIMPL program components are removed.

### The ‘C’ Receiver

```

1 // 'C' receiver: program called c_receiver
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <simpl.h>
7
8 int main()
9 {
10 char *sender;
11 int inNumber;
12 int outNumber;
13 int size = sizeof(int);                                // R1
14 char *me = "RECEIVER";
15
16 // perform simpl name attach
17 if (name_attach(me, NULL) == -1)                      // R2
18 {
19     printf("%s: cannot attach name=%s\n", me, whatsMyError());
20     exit(-1);
21 }
22
23 while (1)
24 {
25     // receive incoming messages
26     if (Receive(&sender, &inNumber, size) == -1)          // R3

```

```

27  {
28    printf("%s: Receive error-%s\n", me, whatsMyError());
29    continue;
30  }
31
32 // calculate square of sent number
33 outNumber = inNumber * inNumber;                                // R4
34
35 // reply squared number to sender
36 if (Reply(sender, &outNumber, size) == -1)                      // R5
37 {
38   printf("%s: Reply error-%s\n", me, whatsMyError());
39   continue;
40 }
41
42 name_detach();                                                 // R6
43 return(1);
44
45 }
```

lines 3-5 Include required ‘C’ header files.

line 6 Include SIMPL header file.

lines 10-14 Variable declarations. Note the receiver program SIMPL name.

lines 17-21 The program performs a ***name\_attach***. Recall that this is required prior to any other SIMPL library calls. The first argument in the function call is the SIMPL name chosen to be unique on the local host. The second argument which in our case is NULL, is a pointer to a user-defined function that would be run at the program’s exit. Again, recall the phone book analogy; this is like adding the receiver’s name to the phone book.

lines 23-41 This is an infinite loop allowing messages to be received, processed and replied to continuously. Many receiver type programs have this format.

lines 26-30 This is where the program receives incoming messages. The first argument of the ***Receive*** call stores the sender’s unique SIMPL identification for later use in the ***Reply***. The second argument is a pointer to memory where the incoming integer will be copied. The third argument dictates the maximum size of the incoming message in order to prevent any overrun of memory space.

line 33 The incoming integer is squared.

lines 36-40 The **Reply** call returns the result of the squaring process to the sender. The first argument represents the sender's SIMPL identification which was gotten from the earlier **Receive** call, the second argument is a pointer to the memory containing the squared integer and the third argument is size of the replied message.

line 43 The SIMPL program components are removed.

### The Java Receiver

```

1 // Java receiver: program called J_receiver.java
2
3 class J_receiver
4 {
5     public static void main(String arg[])
6     {
7         int ret;
8         int sender;
9         int inNumber;                                // R1
10        int outNumber;
11
12        // default constructor
13        Jsimpl simpl = new Jsimpl();
14
15        // perform simpl name attach
16        ret = simpl.nameAttach("RECEIVER");           // R2
17        if (ret == -1)
18        {
19            System.out.println("name attach error");
20            System.exit(-1);
21        }
22
23        while (true)
24        {
25            // receive incoming messages
26            sender = simpl.Receive();                  // R3
27            if (sender == -1)
28            {
29                System.out.println("receive error");
30                System.exit(-1);
31            }
32
33            // unpack the incoming message from the sender
34            inNumber = simpl.unpackInt(simpl.SIMPL_BIN);
35
36            // calculate square of sent number

```

```
37     outNumber = inNumber * inNumber; // R4
38
39     // pack the outgoing reply to the sender
40     simpl.packInt(outNumber, simpl.SIMPL_BIN);
41
42     // reply squared number to the sender
43     ret = simpl.Reply(sender); // R5
44     if (ret == -1)
45     {
46         System.out.println("reply error");
47         System.exit(-1);
48     }
49 }
50 }
51 }
```

line 3 Java class definition.

line 5 Program starting point.

line 13 Simpl instance declaration.

lines 16-21 The program performs a **nameAttach**. Recall that this is required prior to any other SIMPL library calls.

lines 23-53 This is an infinite loop allowing messages to be received, processed and replied to continuously.

lines 26-31 This is where the program receives incoming messages. The **Receive** method returns the sender's unique SIMPL identification for later use in the **Reply**.

line 34 The incoming integer to be squared is retrieved from the incoming message via the unpackInt() method.

line 37 The incoming integer is squared.

line 40 The squared integer is loaded into the reply message via the packInt() method.

lines 43-48 The **Reply** method returns the result of the squaring process to the sender. The first argument represents the sender's SIMPL identification which was gotten from the earlier **Receive** call.

#### 4.4.2 The Senders

##### The Java Sender

```

1 // Java sender: program called J-sender.java
2
3 class J_sender
4 {
5     public static void main( String arg [] )
6     {
7         int ret;
8         int receiver;
9         int inNumber;
10        int outNumber;
11
12        // default constructor
13        Jsimpl simpl = new Jsimpl();
14
15        // perform simpl name attach
16        ret = simpl.nameAttach("SENDER");                                // S2
17        if (ret == -1)
18        {
19            System.out.println("name attach error");
20            System.exit(-1);
21        }
22
23        // name locate the receiver program
24        receiver = simpl.nameLocate("RECEIVER");                          // S3
25        if (receiver == -1)
26        {
27            System.out.println("name locate error");
28            System.exit(-1);
29        }
30
31        // build message and send to receiver
32        for (outNumber = 1; outNumber <= 10; outNumber++)                // S1
33        {
34            // pack the number to be sent into the outgoing message
35            simpl.packInt(outNumber, simpl.SIMPL_BIN);                      // S4
36
37            ret = simpl.Send(receiver);                                     // S5
38            if (ret == -1)
39            {
40                System.out.println("send error");
41                System.exit(-1);
42            }

```

```

43
44     // unpack the replied number from the incoming message
45     inNumber = simpl.unpackInt(simpl.SIMPL_BIN);           // S6
46
47     // print out the numbers
48     System.out.println("num=" + outNumber + " sqr=" + inNumber);
49 }
50
51     ret = simpl.nameDetach();                                // S7
52 }
53 }
```

line 3 Java class definition.

line 5 Program starting point.

line 13 Simpl instance declaration.

lines 16-21 The program performs a ***nameAttach***. Recall that this is required prior to any other SIMPL library calls.

lines 24-29 The program performs a ***nameLocate*** on the receiver program in order to initiate SIMPL communications.

lines 32-52 This is a loop allowing messages to be composed, sent, and processed.

line 35 The integer to be squared is loaded into the outgoing message via the packInt() method.

lines 37-42 The message is sent to the receiver program.

line 45 The squared integer is retrieved from the reply message via the unpackInt() method.

line 51 The SIMPL program components are removed.

### The Python Sender

```

1 # Python sender: program called py-sender.py
2
3 #! /usr/bin/python
4
5 # import necessary modules
6 import sys
7 from csimpl import *
```

```

8  # initialize necessary variables
9  me = "SENDER"
10
11
12 # perform simpl name attach
13 if nameAttach(me, 1024) == -1:                      # S1/S2
14     print("%s: name attach error-%s" %(me, whatsMyError()))
15     sys.exit(-1)
16
17 # name locate the receiver program
18 receiverId = nameLocate("RECEIVER")                  # S3
19 if receiverId == -1:
20     print("%s: name locate error-%s" %(me, whatsMyError()))
21     sys.exit(-1)
22
23 # send messages to RECEIVER
24 for outNumber in range(1, 11):
25     # pack the outgoing message
26     packInt(outNumber, SIMPL_BIN)                      # S4
27
28 # send the message to RECEIVER
29 if send(receiverId) == -1:                          # S5
30     print("%s: send error-%s" %(me, whatsMyError()))
31     sys.exit(-1)
32
33 # unpack the result
34 inNumber = unpackInt(SIMPL_BIN)                      # S6
35
36 # print out the result
37 print("out number=%d in number=%d" %(outNumber, inNumber))
38
39 nameDetach()                                         # S7

```

line 3 Invoke the Python interpreter.

lines 6-8 Import necessary Python and SIMPL modules.

line 11 Set unique SIMPL name.

lines 14-16 The program performs a ***nameAttach***. Recall that this is required prior to any other SIMPL library calls. The argument in the ***nameAttach()*** call is the SIMPL name chosen to be unique on the local host. Again, recall the phone book analogy; this is like adding the sender's name to the phone book. The 1024 is the maximum size that a sent or replied message may be in bytes.

lines 19-22 The sender needs to name locate the receiver program so the various FIFO and shared memory connections can be made. In our phone book analogy this is the same as looking up the phone number in the phone book.

lines 25-41 A loop of ten iterations is run which sends the numbers 1-10 to the receiver, retrieves a reply from the receiver in the form of the square of the sent number and then prints the sent and replied numbers to stdout.

line 27 The number to be squared is packed into the outgoing message.

lines 30-32 The ***send*** call is the mechanism for sending out the number to be squared and retrieving the result from the receiver. The argument represents the receiver's SIMPL ID which was gotten from the earlier ***name-Locate*** call.

line 35 The squared number is unpacked from the replied message.

line 38 The results are printed to the screen.

line 40 The SIMPL program components are removed.

### The ‘C’ Sender

```

1 // 'C' sender: program called c-sender
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <simpl.h>
7
8 int main()
9 {
10    int receiver;
11    int inNumber;
12    int outNumber;
13    int size = sizeof(int);           // S1
14    char *me = "SENDER";
15
16    // perform simpl name attach
17    if (name_attach(me, NULL) == -1)      // S2
18    {
19        printf("%s: cannot attach name=%s\n", me, whatsMyError());
20        exit(-1);
21    }

```

```

22 // name locate the receiver program
23 receiver = name_locate("RECEIVER"); // S3
24 if (receiver == -1)
25 {
26     printf("%s: cannot locate receiver-%s\n", me, whatsMyError());
27     exit(-1);
28 }
29
30
31 // build message and send to receiver
32 for (outNumber = 1; outNumber <= 10; outNumber++) // S4
33 {
34     if (Send(receiver, &outNumber, &inNumber, size, size) == -1)// S5
35     {
36         printf("%s: cannot send to receiver-%s\n", me, whatsMyError());
37         exit(-1);
38     }
39     // print out the messages
40     printf("out number=%d in number=%d\n", outNumber, inNumber); // S6
41 }
42
43 name_detach(); // S7
44 return(1);
45 }
```

lines 3-5 Include required ‘C’ header files.

line 6 Include SIMPL header file.

lines 10-14 Variable declarations. Note the sender program SIMPL name.

lines 17-21 The program performs a ***name\_attach***. Recall that this is required prior to any other SIMPL library calls. The first argument in the function call is the SIMPL name chosen to be unique on the local host. The second argument which in our case is NULL, is pointer to a user-defined function that would be run at the program’s exit. Again, recall the phone book analogy; this is like adding the sender’s name to the phone book.

lines 21-24 The sender needs to name locate the receiver program so the various FIFO and shared memory connections can be made. In our phone book analogy this is the same as looking up the phone number in the phone book.

lines 32-41 A loop of ten iterations is run which sends the numbers 1-10 to the receiver, retrieves a reply from the receiver in the form of the square of the sent number and then prints the sent and replied numbers to stdout.

lines 34-38 The ***Send*** call is the mechanism for sending out the number to be squared and retrieving the result from the receiver. The first argument represents the receiver's SIMPL ID which was gotten from the earlier ***name\_locate*** call, the second argument is a pointer to the memory containing the number to be sent to the receiver, the third argument is a pointer to the memory that will contain the reply from the receiver, the fourth argument is the size in bytes of the second argument and finally, the fifth argument is the size of the reply message from the receiver in bytes.

line 43 The SIMPL program components are removed.

### The Tcl Sender

```

1 # Tcl sender: program tcl_sender.tcl
2
3#!/usr/bin/tclsh
4
5# initialize variables
6set myName SENDER
7lappend auto_path $env(SIMPLHOME)/lib
8package require Fctclx
9
10# perform simpl name attach
11set myslot [name_attach $myName] ;# S2
12if { [string compare $myslot "NULL"] == 0 } {
13    puts stdout [format "%s: cannot attach name" $myName]
14    exit
15}
16
17# name locate the receiver program based on its SIMPL name
18set recvID [name_locate "receiver"] ;# S3
19if {$recvID == -1} {
20    puts stdout [format "%s: cannot locate receiver" $myName]
21    name_detach
22    exit
23}
24
25# compose and Send message; retrieve reply and display
26for {set outNumber 1} { $outNumber <= 10} {incr outNumber} { ;# S4
27    set sMsg [binary format "i1" $outNumber]
28    set sBytes [string length $sMsg]
```

```

29 set rMsg [Send $recvID $sMsg $sBytes] ;# S5
30 binary scan $rMsg i1i1 fromWhom rBytes
31 if { $rBytes == -1} {
32     puts stdout [format "%s: cannot send to receiver" $myName]
33     break
34 }
35
36 binary scan $rMsg x8i1 inNumber ;# S6
37 puts stdout [format "out number = %d in number = %d" \
38     $outNumber $inNumber]
39 }
40
41 name_detach ;# S7
42 exit

```

line 3 Invoke the Tcl interpreter.

line 6 Set unique SIMPL name.

lines 7-8 Include SIMPL library package.

lines 11-15 The program performs a ***name\_attach***. Recall that this is required prior to any other SIMPL library calls. The argument in then ***name\_attach*** call is the SIMPL name chosen to be unique on the local host. Again, recall the phone book analogy; this is like adding the receiver's name to the phone book.

lines 18-23 The sender needs to name locate the receiver program so the various FIFO and shared memory connections can be made. In our phone book analogy this is the same as looking up the phone number in the phone book.

lines 26-39 A loop of ten iterations is run which sends the numbers 1-10 to the receiver, retrieves a reply from the receiver in the form of the square of the sent number and then prints the sent and replied numbers to stdout.

lines 29 The ***Send*** call is the mechanism for sending out the number to be squared and retrieving the result from the receiver. The first argument represents the receiver's SIMPL ID which was gotten from the earlier ***name\_locate*** call, the second argument is a pointer to the memory containing the number to be sent to the receiver, the third argument is the size in bytes of the second argument. The procedure returns the replied message.

lines 30-38 The reply message is extracted and displayed.

line 41 The SIMPL program components are removed.

#### 4.4.3 What's In a Name?

Suppose that you wanted all of the senders above to be able to send to one of the receivers at the same time. This is actually what SIMPL is normally all about. For the sake of illustration let's use the Tcl receiver. Its SIMPL name is *receiver*. We want to run the Tcl, Python, and 'C' senders at the same time. The problem is that in the sample code they all have the same hard-coded SIMPL name, namely *sender*. We know that SIMPL names must be unique per host. No problem here, we simply change each program's SIMPL name to something different. For example, set the SIMPL names to T\_sender, P\_sender, and C\_sender for the Tcl, Python, and 'C' sending programs respectively. Now all of the sender programs can be run simultaneously and communicate with whichever receiver program desired.

The receiver programs as they stand would be in conflict with their SIMPL names if they were to be run together. If their hard-coded names were changed to make them unique, the senders could then choose which receiver they would send their message to. Most often SIMPL programmers don't hard-code SIMPL names. They elect instead to have the SIMPL name passed into the program via the command line interface. For SIMPL senders, both its own name and the name of the receiver would be command line parameters.

Let's take a look at an example of how this might work. Suppose that we want to have the python sender program to be able to send its message to any of the three receiver programs. The three receiver programs could take their unique SIMPL names from the command line. Let's call the Tcl, Python, and 'C' receivers T\_receiver, P\_receiver, and C\_receiver respectively. We could use a shell script to automate the startup of our various programs. Such a script might look like the following:

#### Startup Script

```
1 # Startup Script: program called startup
2
3#!/usr/bin/bash
4
5# start the Tcl receiver program
6/usr/bin/wish tcl-receiver.tcl T_receiver &
7
```

```
8 # start the Python receiver program
9 /usr/bin/python python_receiver.py P_receiver &
10
11 # start the 'C' receiver program
12 c_receiver C_receiver &
13
14 # start the Python sender program
15 /usr/bin/python python_sender.py $1
```

line 3 Invoke the Shell interpreter.

line 6 Invoke the Tcl interpreter to run the tcl\_receiver.tcl script in the background. Note the string *T\_receiver* on the command line. This will be the SIMPL name of the tcl\_receiver.tcl program.

line 9 Invoke the Python interpreter to run the python\_receiver.py script in the background. Note the string *P\_receiver* on the command line. This will be the SIMPL name of the python\_receiver.py program.

line 12 Run the c\_receiver binary in the background. Note the string *C\_receiver* on the command line. This will be the SIMPL name of the c\_receiver program.

line 15 Invoke the python interpreter to run the python\_sender.py script in the foreground. Note the *\$1* on the command line. This will be how we would pass the name of the receiver we want python\_sender.py to send its message to. Observe the following list of possibilities:

1. *startup T\_receiver* would have the python\_sender.py program send its message to tcl\_receiver.tcl.
2. *startup P\_receiver* would have the python\_sender.py program send its message to python\_receiver.py.
3. *startup C\_receiver* would have the python\_sender.py program send its message to c\_receiver.

## 4.5 Return Codes

We note in section 4.4 that the various SIMPL library function calls had return values. In all of these cases a -1 represents a failure of some sort. Another

SIMPL library function was used to provide a string representation of an error condition in the same way that strerror(errno) might be used; the name of this function is ***whatsMyError***.

In the case of the receiver program in section 4.4, the only possible fatal error resulting in a program exit was related the ***name\_attach*** call. This is fatal because without a ***name\_attach*** no further processing is possible. But note the ***Receive*** and ***Reply*** calls, both simply continue in the loop in the case of a failure. The failure might simply have to do with one sender program and if the receiver exits, then no other sender will be able to find it.

In the case of the sender program, all failures were considered fatal because there would be no need to carry on after that point in the case of an error. SIMPL errors should be treated by the programs using SIMPL on a case by case basis. If a sender program becomes unreachable between the receiver receiving the message and the reply being sent back, an error on the reply will occur. This does not indicate a failure in the receiver's situation; the problem may have occurred due to a dropped line between a remote sender and a local receiver for example.

See appendices C and O for more detailed information.

#### 4.5.1 Warnings and Errors

Core SIMPL functions report errors via return codes. It is up to the user to flag and deal with them in an expedient fashion. However, SIMPL does little more than return a -1 indicating a problem. Warnings and errors which are internal to SIMPL are date and time stamped and written to a rolling log file. This file has a maximum size of 100KB. If this size is exceeded, the file is simply written over and started again. It is often a good place to look for clues to problems. Currently, this file is called *simpl* and lives in the */var/tmp* directory on a Linux-based system.

## 4.6 Security

The only aspect of messaging that is of interest to SIMPL is the size of the message, *period*. On a number of occasions, as the SIMPL project matured, suggestions were made to add encryption and various security measures to the core of SIMPL. We've resisted because it is our belief that this is the wrong layer in which to put security measures. The SIMPL application developer is always able to design an algorithm which would build a message, encrypt it,

send it, receive it, decrypt it and then carry on with business. In this way the encryption algorithm is entirely in the application layer. Moreover, there is no form of sniffing embedded within SIMPL. If one needed to sniff messages, there are far better ways to do it.

Finally, there is a veracity issue here. SIMPL is as advertised. The code is fully open for all to inspect. There are no back doors buried in SIMPL that reports your messages to any outside party. If such a thing gets added by a third party, it goes against the express wishes of the authors and has been done utterly without their consent. The mainline SIMPL software available on the website is secure from any of this sort of tampering. If back doors exist on someone's system then they have been added after the fact.

## 4.7 Installing SIMPL

By this point you may be anxious to try out some things with SIMPL. Appendix A describes how to get started. There is also lots of valuable information on the main SIMPL website<sup>8</sup> along with helpful persons on the SIMPL mailing list.

## 4.8 Summary

The core SIMPL library can be summarized as:

- A very clean API of five main functions: *name\_attach*, *name\_locate*, *Send*, *Receive*, and *Reply*.
- A sandbox coordinated by the FIFO\_PATH environment variable where all the FIFOs responsible for message synchronization actually reside.

## 4.9 Next Chapter

In the next chapter we are going to take a look at the messages that are sent between programs using SIMPL. In particular, we are going to examine what is known as tokenized message passing and how it is used to make a clean and functional interface between sending and receiving programs.

---

<sup>8</sup>The main SIMPL website is at: <http://www.icanprogram.com/simpl>.

# Chapter 5

## Messages

”message n. communications sent”<sup>1</sup>

### 5.1 Messages and Messaging

A SIMPL message is contained in a contiguous block of bytes. A SIMPL transaction involves the transportation of this SIMPL message from one SIMPL module to another. The information in that contiguous block of bytes will have both content (actual field values) and format (field layout and field types). The content in a SIMPL message must be packed by the initiator and unpacked by the recipient. The format is subject to a prior design exercise, the result of which must be communicated to both modules involved in a SIMPL transaction.

In chapter 4 the SIMPL core library was discussed. The core library concerns itself with the transport of a SIMPL message from one local SIMPL module to another. The content or structure associated with that contiguous block of bytes is of no concern to the core library. Only the size of the contiguous block of bytes matters to the core library.

In chapter 3 the exercise of SIMPL application design was discussed. The design of the SIMPL message format is equally important to the success of a SIMPL application. This chapter is about the *art of SIMPL message design*.

---

<sup>1</sup>Collins English Dictionary

## 5.2 Messages

There are many ways to structure a contiguous block of bytes into a SIMPL message. In all cases this involves laying out and specifying the various pieces of information and then arranging them in a specific order. In all cases the SIMPL communicator (sender, replier) *packs* the message and the SIMPL recipient (receiver, blocked sender) *unpacks* the message.

The structural information embodied in these pack/unpack operations relies on an agreement between the communicating parties as to the message structure. Once a SIMPL message has been created, it is stored and transmitted as a block of shared memory owned by the sender. While not commonly used, SIMPL ‘C’ programs can manipulate the message directly in the shared area without needing to copy the message to a local buffer. All the other SIMPL-supported languages require that the message be block copied in and out of shared memory and manipulated in a local buffer. This is commonly the way ‘C’ algorithms operate as well.

SIMPL message designs can be roughly classified into two schemes:

1. Binary based messages.
2. Text based messages.

As a rough rule of thumb, binary schemes are the most efficient while text based schemes are the most adaptable. SIMPL doesn’t impose any particular messaging scheme, but rather the application architecture and deployment environment does.

A binary based message is often very convenient for ‘C’ programs because the concept of *binary structures* is readily available within ‘C’. However, binary primitives such as integers are affected by host computer architecture (32/64 bit, endians). This alters the size and alignment of these primitives within a binary message. While not a problem for local message exchange or network message exchange on homogeneous hardware, it does impose added challenges in heterogeneous networks. Binary messages are not *natural* fits to the non-‘C’ languages supported by SIMPL (Python, Java, and Tcl). While ‘C’ programs can readily pack and unpack binary messages by employing a ‘C’ structure pointer to access the fields, non-‘C’ languages must pay closer attention to their packing and unpacking alignment and sizing issues.

Text messages, on the other hand, are not challenged by architecture or byte ordering/alignment issues. The primitive in a text message is typically a single byte which is atomic on all systems. However, text messages are more

computationally intensive to pack and unpack than their binary equivalents. To transmit equivalent content, text messages tend to be larger than their binary equivalents.

Within the context of a binary or a text message there are many possible design strategies. One of the most powerful is called tokenized messaging. Tokenized messaging divides the message into a common part of fixed size and a message specific part of variable size. The common part is unpacked and parsed first. This in turn determines the structure of the message specific part. Tokenized messages are powerful because they are so readily extensible. Binary tokenized messages, where the common element is a single atomic primitive, are exceedingly easy to parse, and as such are very popular amongst ‘C’ SIMPL developers.

For text messages another popular strategy is to utilize punctuation characters to separate fields. The simplest amongst those strategies is called *tag: value* pairing. These message designs tend to evolve very rigid syntax rules to keep the parsing simpler. XML based messaging is also used to structure text messages. XML messages have flexible syntax rules. While XML is very powerful, it tends to be computationally expensive to unpack and decompose.

For a SIMPL communication to be successful the block of bytes needs to be transmitted along with the structural information about how to interpret those bytes. Most SIMPL applications don’t actually *transmit* this structural information, but rather rely on a prior *agreement* about that structure and transmit instead maps or indexes into those prior *agreements*. Such messaging schemes are said to be non-self-describing. XML message structures, on the other hand, can transmit structural information alongside the actual blob of bytes and are said to be self-describing.<sup>2</sup>

A SIMPL application typically consists of many different SIMPL modules which intercommunicate. Each communication pathway could have a different messaging scheme. However, for ease of documentation and maintenance a single messaging scheme is often chosen to be application wide.

The art of choosing a SIMPL message design strategy is a complex interconnected optimization. The original problem decomposition into SIMPL modules is often iteratively coupled to SIMPL message design. The hardware and network architecture chosen for the solution plays a role. The choice of programming language associated with a given SIMPL module is then coupled to all of the above. Furthermore, which programming languages are chosen are often a matter of

---

<sup>2</sup>In practice, even XML messages rely on prior agreements about how tags are to be interpreted.

expediency and personal taste. For example, it is more expedient to write a GUI in Tcl/Tk or Python/Tk or Java/Swing than in ‘C’/X Windows. To complicate the situation even further, the choice of message design affects the overall SIMPL application performance and cost of maintenance and expansion.

## 5.3 Using Messages

Once a particular SIMPL message has been designed, that is, whether the message is binary/text, the data content and order of the data items have all been determined, then the message has to be constructed (packed) by a sender and deconstructed (unpacked) by a receiver.

The procedures used to pack and unpack messages are language dependent. Due to the differences inherent in the various programming languages which support SIMPL libraries, creating messages in each of these languages is done somewhat differently.

### 5.3.1 Packing and Unpacking Messages

Packing refers to the process of composing a structured block of bytes in an outgoing message. Unpacking refers to the inverse process of decomposing a structured block of bytes upon reception. Depending on the choice of the SIMPL supported programming language employed and the choice of message design, the packing and unpacking processes can be of varying computational intensity. While not universally true in all message designs, the unpacking process is often the more computationally intensive.

Binary messages are often used in ‘C’/SIMPL programs due to their natural fit within the terms of the language. Binary messages in ‘C’ are particularly easy to pack and unpack as illustrated in the example below.

In the example, a sender-receiver pair collaborate to calculate the periodic payments on an amortized loan. The sender initiates the computation by transmitting loan parameters. The receiver computes the result and returns it back to the sender. Since the various message concepts are adequately illustrated in the receiver code the sender listing has been omitted. This example is carried through the chapter.<sup>3</sup>

---

<sup>3</sup>Note that in the examples the normal error checking code has been removed for the sake of clarity and brevity.

receiver1.c

```
1 // receiver1.c
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <simpl.h>
6 #include <math.h>
7
8 typedef struct
9 {
10     double principal; // initial loan amount (principal)
11     double rate;      // interest rate per period
12     int numPeriods;   // total number of payment periods
13 } PAYMENT;
14
15 typedef struct
16 {
17     double result; // result of calculation
18 } RESULT;
19
20 char inMem[1024]; // memory for received messages
21 char outMem[1024]; // memory for replied messsages
22
23 int main()
24 {
25     const char *me = "RECEIVER";
26     char *sender;
27     double payment, p, r;
28     int n;
29     PAYMENT *in;
30     RESULT *out;
31
32     // SIMPL name attach
33     name_attach(me, NULL);
34
35     while (1)
36     {
37         // receive incoming SIMPL messages
38         Receive(&sender, inMem, 1024);
39
40         // line up PAYMENT message structure with incoming message memory
41         in = (PAYMENT *)inMem;
42
43         // unpack the incoming message
44         p = in->principal;
45         r = in->rate;
```

```
46     n = in->numPeriods;  
47  
48     // calculate the payment amount per period  
49     payment = p * r * pow(1 + r, n) / ( pow(1 + r, n) - 1);  
50  
51     // line up payment result with the outgoing message memory  
52     out = (RESULT *)outMem;  
53  
54     // pack the reply  
55     out->result = payment;  
56  
57     // reply to sender  
58     Reply(sender, outMem, sizeof(RESULT));  
59 }  
60  
61 return(0);  
62 }
```

lines 3-6 Required header files.

lines 8-13 The definition of the message that is to be received by this program encapsulated in a structure format. We call a message of this type *PAYMENT*.

lines 15-18 The definition of the reply message that is to be returned back to the sending program encapsulated in a structure format. We call a message of this type *RESULT*.

lines 20-21 Global incoming and outgoing message buffer declarations. The incoming messages (Receive) are stored in the inMem buffer where they are later unpacked according to the particular message design and processed. The outgoing messages (Reply) are packed in the outMem buffer where they are sent back to the sender.

line 33 The always required SIMPL name attach.

lines 55-139 An infinite loop wherein messages are received, information unpacked from the incoming message, calculations made, calculated data packed into reply messages, and reply messages returned back to the sending program.

line 38 Messages are received and are contained in the memory buffer called inMem.

line 41 A pointer of message type *PAYMENT* is directed at the memory buffer called *inMem*. This can be thought of as a template overlaying memory in order to make sense out of the memory contents.

lines 44-46 With the *PAYMENT* message template in place, the various message data items are unpacked.

line 49 The amortized payment is calculated for each payment period.

line 52 A pointer of message type *RESULT* is directed at the memory buffer called *outMem*. This is where the outgoing message will reside.

line 55 The calculated payment is packed into the reply message.

line 58 The reply message is sent back to the sending program.

A new data type has been invented, **PAYMENT**, to structure a SIMPL message. The **typedef struct** construct tells the ‘C’ compiler how this new data type is laid out in memory. However, the **typedef struct** doesn’t actually allocate any memory. Notice that the **PAYMENT** structure is never used to declare a variable. It is used to declare a pointer which is then mapped onto a contiguous block of memory (*inMem*). The other thing to notice is that the SIMPL API functions (**Receive()** and **Reply()**) only deal with the contiguous chunks of memory (*inMem* and *outMem*) and never with message structure itself. This separation of duty is what leads to all the power in the SIMPL messaging.

In several non-‘C’ languages supported by SIMPL (Python, Java) the packing and unpacking of binary messages are accomplished via a family of methods or functions. These methods use a block of working memory *hidden* from the programmer. This memory is generally comprised of an *inBuffer* and an *outBuffer*. The *inBuffer* temporarily stores incoming messages. The unpacking routines operate on this buffer to extract the various data items from the incoming message. The *outBuffer* temporarily stores outgoing messages. The packing routines operate on this buffer to structure the various data items in an outgoing message. In the case of Java, the sizes of these buffers are set automatically to default values although these can be changed via the constructors. The same also applies to Python when the packing/unpacking module is used.

The following example has been written in Python and is a direct translation of the preceding ‘C’ program example. Note the use of explicit function calls for the packing and unpacking of messages.

*receiver1.py*

```
1 # receiver1.py
2
3 from csimpl import *
4
5 # SIMPL name attach
6 nameAttach("RECEIVER", 1024)
7
8 while 1:
9     # receive incoming SIMPL messages
10    messageSize, senderId = receive()
11
12    # unpack the incoming message
13    p = unpackDouble(SIMPL_BIN)
14    r = unpackDouble(SIMPL_BIN)
15    n = unpackInt(SIMPL_BIN)
16
17    # calculate the payment amount per period
18    payment = p * r * ((1 + r) ** n) / (((1 + r) ** n) - 1)
19
20    # pack the reply
21    packDouble(payment, SIMPL_BIN)
22
23    # reply to sender
24    reply(senderId)
```

line 3 The SIMPL function calls and packing/unpacking functions are imported from the csimpl module.

line 6 The always required SIMPL name attach. Note that the program's SIMPL name is "RECEIVER" and that the largest allowable message is 1 kbyte.

lines 8-27 An infinite loop wherein messages are received, information unpacked from the incoming message, calculations made, calculated data packed into reply messages, and reply messages returned back to the sending program.

line 10 Messages are received. The messages are contained within a buffer that is defined within the csimpl module.

lines 13-15 The various message data items are unpacked.

line 18 The amortized payment is calculated for each payment period.

line 21 The calculated payment is packed into the reply message.

line 24 The reply message is sent back to the sending program.

Note the difference in *code size* between the ‘C’ program (`receiver1.c`) and the Python equivalent (`receiver1.py`); the ‘C’ program appears to be much longer. The reason for this is that some of what the ‘C’ program does explicitly has been buried within the `csimpl` module function calls.

### 5.3.2 Tokenized Binary Messages

A particularly powerful subset of binary messages are known as tokenized binary messages. They are widely used because they are also one of the simplest (see Figure 5.1). Tokenized text messaging is also possible and confers many of the same advantages as the binary counterpart.

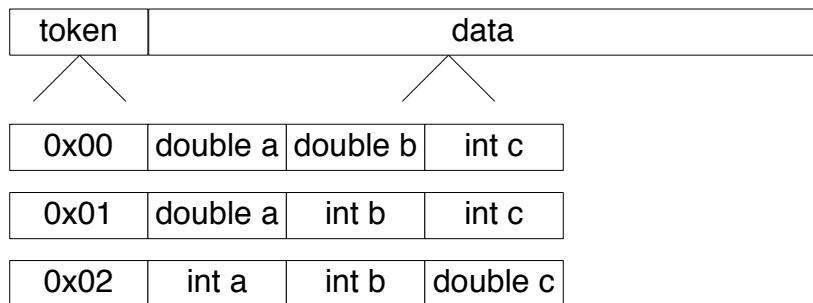


Figure 5.1: SIMPL Tokenized Message

Imagine this collection of bytes is divided into two parts. The first field of bytes which is of fixed length is always present in every message and when taken together, these bytes represent a unique message identifier called a token. In Figure 5.1 the token is a single binary word and can take on three values **0x00**, **0x01** and **0x02**. The token is always unpacked and parsed first. The balance of the message (which can be of variable length) represents the token context-sensitive data. When two processes wish to exchange such messages they simply agree on a token scheme and on the format for each tokenized message they want to exchange. The token represents an index (or map) into that agreed upon message formating scheme. To the SIMPL interprocess message transport layer all this is still just a contiguous collection of bytes. ie. SIMPL transports the messages, the applications compose and interpret them.

Here is a helpful analogy to illustrate tokenized messaging. When the first dedicated wordprocessor machines were introduced they looked something like a modern PC with a CPU box, a keyboard, and a monitor screen. Unlike modern word processor packages, the software on these machines came on and ran from a series of floppy diskettes. Along with each diskette came a set of cardboard templates which could be laid down over the keyboard function keyset. The template had holes where the keys could poke through and relabelled each of the keys in a more friendly way for the particular piece of wordprocessing software that was running.

If the function keys on that old dedicated wordprocessor keyboard are the analogue to bytes in our SIMPL message, tokenized messages are the software analogue of that cardboard template.

If two processes exchanged a fixed structure of data, as in the previous example code, there wouldn't be a need for tokenizing the message. Both processes would agree on the message format which captured that fixed structure. However, it is a rare software problem that can be decomposed into a single data structure (or if attempted this the structure rapidly becomes unwieldy). Furthermore, most software is extended as new features are added. This is where tokenized messaging really shines. Tokenized messaging is a software framework which, when applied to a contiguous block of bytes, promotes code reuse, and code extendability.

The first example to illustrate the concept of tokenized binary messaging will use the ‘C’ language. While the concept of a tokenized binary message lends itself particularly well to that language, tokenized messaging is by no means restricted to ‘C’. SIMPL developers regularly construct, exchange and decompose tokenized messages in Tcl, Java, and Python: the other principal languages with which SIMPL modules are created.

The code from the first example called receiver1.c has been augmented for this example - receiver2.c. The code has been changed only with respect to adding the notion of tokenized messaging. Only the changes will be discussed because both programs are functionally identical.

receiver2.c

```
1 // receiver2.c
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <simpl.h>
6 #include <math.h>
7
8 typedef enum
9 {
10     PAYMENT_AMOUNT_PER_PERIOD=0
11 } EXAMPLE_TOKENS;
12
13 typedef struct
14 {
15     int token;      // message token
16     double principal; // initial loan amount (principal)
17     double rate;    // interest rate per period
18     int numPeriods; // total number of payment periods
19 } PAYMENT;
20
21 typedef struct
22 {
23     int token;      // message token
24     double result;   // result of calculation
25 } RESULT;
26
27 int paymentAmountPerPeriod(void);
28
29 char inMem[1024]; // memory for received messages
30 char outMem[1024]; // memory for replied messsages
31
32 int main()
33 {
34     const char *me = "RECEIVER";
35     char *sender;
36     int *inToken;
37     int outSize;
38
39     // SIMPL name attach
40     name_attach(me, NULL);
41
42     // line up the token with the beginning of the message memory
43     inToken = (int *)inMem;
44
45     while (1)
```

```
46  {
47  // receive incoming SIMPL messages
48  Receive(&sender, inMem, 1024);
49
50 // react accordingly to the value of the token
51 switch (*inToken)
52 {
53 case PAYMENT_AMOUNT.PERIOD:
54     outSize = paymentAmountPerPeriod();
55     break;
56
57 default:
58     printf("%s: Unknown token.\n", me);
59     continue;
60 }
61
62 // reply to sender
63 Reply(sender, outMem, outSize);
64 }
65
66 return(0);
67 }
68
69 int paymentAmountPerPeriod()
70 {
71 double payment, p, r;
72 int n;
73 PAYMENT *in;
74 RESULT *out;
75 // inMem is global
76 // outMem is global
77
78 // line up PAYMENT message structure with incoming message memory
79 in = (PAYMENT *)inMem;
80
81 // unpack the incoming message
82 p = in->principal;
83 r = in->rate;
84 n = in->numPeriods;
85
86 // calculate the payment amount per period
87 payment = p * r * pow(1 + r, n) / (pow(1 + r, n) - 1);
88
89 // line up payment result with outgoing message memory
90 out = (RESULT *)outMem;
91
```

```
92 // pack the reply
93 out->token = PAYMENT_AMOUNT_PER_PERIOD;
94 out->result = payment;
95
96 return ( sizeof(RESULT) );
97 }
```

lines 8-11 An enumeration is used to define and give a unique value to any tokens that are used in the message exchange. The programmer need only choose a unique and hopefully descriptive label for the token, the ‘C’ compiler determines the resulting unique numerical value used within the program.

lines 13-19 The definition of the message that is to be received by this program encapsulated in a structure format. This is the same message definition used above in receiver1.c with the important exception that the first structure element is the value of the token.

lines 21-25 The definition of the reply message that is to be returned back to the sending program encapsulated in a structure format. This is the same message definition used above in receiver1.c with the important exception that the first structure element is the value of a token.

line 43 An integer pointer is aimed at the beginning of the memory buffer that contains an incoming message. This is because the token is the first item in the message. In our case, the token is always at the beginning of the message and is always an integer.

lines 51-60 The switch examines the value of the token and directs the appropriate action accordingly. In our example there is only one token that is considered to be correct. If the token value is not known to the program, the message is ignored.

lines 69-97 The functional part of the program dealing with calculating payments has been encapsulated in a separate function in order to clean up the flow of the program. This style issue will become important in the next example.

In addition to the **PAYMENT** data type an enumerated list of tokens has been created. The **typedef enum** conveniently arranges that token list with the **PAYMENT\_AMOUNT\_PER\_PERIOD** token being the first in that list.

Just like in the cardboard example, there are two steps required in deploying a tokenized binary message in ‘C’:

- first one needs to actually design and create it (or at least unwrap it from the box) - the typedef struct part.
- secondly one needs to lay it down over the function keys - the pointer declaration and pointer instantiation parts.

This code snip forms the basic framework for all SIMPL modules which are written in ‘C’ and use tokenized binary messaging.

The following example written in Python is a translation of receiver2.c.

### *receiver2.py*

```

1 # receiver2.py
2
3 from csimpl import *
4
5 PAYMENT_AMOUNT_PER_PERIOD = 0
6
7 def paymentAmountPerPeriod():
8     # unpack the balance of the message
9     p = unpackDouble(SIMPL_BIN)
10    r = unpackDouble(SIMPL_BIN)
11    n = unpackInt(SIMPL_BIN)
12
13    # calculate the payment amount per period
14    payment = p * r * ((1 + r) ** n) / (((1 + r) ** n) - 1)
15
16    # pack the reply
17    packInt(PAYMENT_AMOUNT_PER_PERIOD, SIMPL_BIN)
18    packDouble(payment, SIMPL_BIN)
19
20    # SIMPL name attach
21    nameAttach("RECEIVER", 1024)
22
23 while 1:
24     # receive incoming SIMPL messages
25     messageSize, senderId = receive()
26
27     # unpack the message token
28     token = unpackInt(SIMPL_BIN)
29
30     if token == PAYMENT_AMOUNT_PER_PERIOD:
31         paymentAmountPerPeriod()
32     else:
33         continue
34

```

```
35 | # reply to sender  
36 | reply(senderId)
```

line 3 The SIMPL function calls and packing/unpacking functions are imported from the csimpl module.

line 5 A variable defining what will be taken as a *token* value is defined.

lines 7-18 A function which will take care of any processing with respect to the token value is defined.

line 21 The always required SIMPL name attach. Note that the program's SIMPL name is "RECEIVER" and that the largest allowable message is 1 kbyte.

lines 23-39 An infinite loop wherein messages are received, information unpacked from the incoming message, calculations made, calculated data packed into reply messages, and reply messages returned back to the sending program.

line 25 Messages are received. The messages are contained within a buffer that is defined within the csimpl module.

line 28 The value of the incoming message token is unpacked.

lines 30-33 The value of the token is checked. The value of the token directs the course of events.

line 31 The appropriate function is called to take the desired course of action.

lines 9-11 The various message data items are unpacked.

line 14 The amortized payment is calculated for each payment period.

lines 17-18 The reply token and the calculated payment is packed into the reply message.

line 36 The reply message is sent back to the sending program.

The examples so far have dealt with trivial messaging. In this simple world one could have gotten away without tokens. Unfortunately, the real world is much more complicated; often programs must deal with a number of messages,

not just one. The use of a token uniquely identifies the message type and then directs actions taken. The ability to extend a tokenized message implementation is why this approach is popular.

In the example below this code has been extended to add two new features: **rate per payment period** and **annual interest rate**.

### receiver3.c

```
1 // receiver3.c
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <simpl.h>
6 #include <math.h>
7
8 typedef enum
9 {
10    PAYMENT_AMOUNT_PER_PERIOD=0,
11    RATE_PER_PAYMENT_PERIOD,
12    ANNUAL_INTEREST_RATE
13 } EXAMPLE_TOKENS;
14
15 typedef struct
16 {
17    int token;          // message token
18    double principal; // initial loan amount (principal)
19    double rate;        // interest rate per period
20    int numPeriods;   // total number of payment periods
21 } PAYMENT;
22
23 typedef struct
24 {
25    int token;          // message token
26    double interest;   // nominal annual interest rate
27    int numCperiods;  // number of compounding periods per year
28    int numPperiods;; // number of payment periods per year
29 } RATE;
30
31 typedef struct
32 {
33    int token;          // message token
34    int numCperiods;  // number of compounding periods per year
35    int numPperiods;; // number of payment periods per year
36    double rate;        // rate per payment period
37 } ANNUAL;
```

```
39 typedef struct
40 {
41     int token;      // message token
42     double result;   // result of calculation
43 } RESULT;
44
45 int paymentAmount PerPeriod(void);
46 int ratePerPaymentPeriod(void);
47 int annualInterestRate(void);
48
49 char inMem[1024]; // memory for received messages
50 char outMem[1024]; // memory for replied messsages
51
52 int main()
53 {
54     const char *me = "RECEIVER";
55     char *sender;
56     int *inToken;
57     int outSize;
58
59 // SIMPL name attach
60 name_attach(me, NULL);
61
62 // line up the token with the beginning of the message memory
63 inToken = (int *)inMem;
64
65 while (1)
66 {
67     // receive incoming SIMPL messages
68     Receive(&sender, inMem, 1024);
69
70     // react accordingly to the value of the token
71     switch (*inToken)
72     {
73         case PAYMENT_AMOUNT_PER_PERIOD:
74             outSize = paymentAmountPerPeriod();
75             break;
76
77         case RATE_PER_PAYMENT_PERIOD:
78             outSize = ratePerPaymentPeriod();
79             break;
80
81         case ANNUAL_INTEREST_RATE:
82             outSize = annualInterestRate();
83             break;
84 }
```

```
85     default:
86         printf("%s: Unknown token.\n", me);
87         continue;
88     }
89
90     // reply to sender
91     Reply(sender, outMem, outSize);
92 }
93
94 return(0);
95 }
96
97 int paymentAmountPerPeriod()
98 {
99     double payment, p, r;
100    int n;
101    PAYMENT *in;
102    RESULT *out;
103    // inMem is global
104    // outMem is global
105
106    // line up PAYMENT message structure with incoming message memory
107    in = (PAYMENT *)inMem;
108
109    // unpack the incoming message
110    p = in->principal;
111    r = in->rate;
112    n = in->numPeriods;
113
114    // calculate the payment amount per period
115    payment = p * r * pow(1 + r, n) / (pow(1 + r, n) - 1);
116
117    // line up payment result with the outgoing message memory
118    out = (RESULT *)outMem;
119
120    // pack the reply
121    out->token = PAYMENT_AMOUNT_PER_PERIOD;
122    out->result = payment;
123
124    return(sizeof(RESULT));
125 }
126
127 int ratePerPaymentPeriod()
128 {
129     double rate, i;
130     int n, p;
```

```
131 RATE *in;
132 RESULT *out;
133 // inMem is global
134 // outMem is global
135
136 // line up RATE message structure with incoming message memory
137 in = (RATE *)inMem;
138
139 // unpack the incoming message
140 i = in->interest;
141 n = in->numCperiods;
142 p = in->numPperiods;
143
144 // calculate the interest rate per payment period
145 rate = pow(1 + i / n, (double)n / (double)p) - 1;
146
147 // line up rate result with the outgoing message memory
148 out = (RESULT *)outMem;
149
150 // pack the reply
151 out->token = RATEPERPAYMENTPERIOD;
152 out->result = rate;
153
154 return(sizeof(RESULT));
155 }
156
157 int annualInterestRate()
158 {
159     double annual, r;
160     int n, p;
161     ANNUAL *in;
162     RESULT *out;
163     // inMem is global
164     // outMem is global
165
166     // line up PAYMENT message structure with incoming message memory
167     in = (ANNUAL *)inMem;
168
169     // unpack the incoming message
170     n = in->numCperiods;
171     p = in->numPperiods;
172     r = in->rate;
173
174     // calculate the annual interest rate
175     annual = n * (pow(r + 1, (double)p / (double)n) - 1);
176 }
```

```
177 // line up rate result with the outgoing message memory
178 out = (RESULT *)outMem;
179
180 // pack the reply
181 out->token = ANNUAL_INTEREST_RATE;
182 out->result = annual;
183
184 return(sizeof(RESULT));
185 }
```

lines 8-13 The enumeration has been extended to include two more tokens.

These new tokens will be used to uniquely identify their respective messages.

lines 23-37 Two more message definitions have been added to the program. Each of these will be identified by the two new tokens defined by extending the enumeration.

lines 71-88 The switch examines the value of the token and directs the appropriate action accordingly. The switch now recognizes the addition of the two new tokens and directs the appropriate action for each of the message types.

lines 127-185 Two new functions have been added to deal with the two new message types. The issue of style was mentioned at this point in the last example. The linearity and readability of the program is maintained by the series of steps: token-message-processing function-reply.

Note what has happened here. With a very few lines of recipe code, two completely new features were added to this module. While these messages are carried in the generic 1K buffers, they have completely different structures. The operative word is **EXTENDABLE**. Extendable is the most valuable attribute to a piece of *real world* software bar-none.

The new feature recipe for the receiver would look like this:

- modify the enum to add a new token,
- add a new message template,
- add a new case statement in the receiver switch,
- recompile and run.

More importantly the probability of breaking existing receiver code with this change is very low because each token is handled in a separate isolated code block. The KISS (Keep It Simple Stupid) philosophy is sadly a lost art amongst programmers.

The extendability of tokenized messages in Java, Tcl, or Python is almost as straightforward.<sup>4</sup>

The following example written in Python is a translation of receiver3.c. The convenience of a ‘C’ enumeration is not exactly supported within Python and so the tokens are explicitly defined as numerical variables. Since the values of the tokens need to be known by at least two separate programs (a sender and a receiver), these values could be contained in a project module. Note that the enumerations used in ‘C’ to define tokens are usually kept in a header file that is available for inclusion by any program requiring those token values.

### *receiver3.py*

```
1 # receiver3.py
2
3 from csimpl import *
4
5 PAYMENT_AMOUNT_PER_PERIOD = 0
6 RATE_PER_PAYMENT_PERIOD = 1
7 ANNUAL_INTEREST_RATE = 2
8
9 def paymentAmountPerPeriod():
10     # unpack the balance of the message
11     p = unpackDouble(SIMPL_BIN)
12     r = unpackDouble(SIMPL_BIN)
13     n = unpackInt(SIMPL_BIN)
14
15     # calculate the payment amount per period
16     payment = p * r * ((1 + r) ** n) / (((1 + r) ** n) - 1)
17
18     # pack the reply
19     packInt(PAYMENT_AMOUNT_PER_PERIOD, SIMPL_BIN)
20     packDouble(payment, SIMPL_BIN)
21
22 def ratePerPaymentPeriod():
23     # unpack the balance of the message
24     i = unpackDouble(SIMPL_BIN)
25     n = unpackInt(SIMPL_BIN)
26     p = unpackInt(SIMPL_BIN)
```

<sup>4</sup>In Tcl, for example, the enum becomes an explicit list of constants and the structure becomes a format string for a **binary scan** or **binary format** statement.

```
27
28     # calculate the interest rate per payment period
29     n *= 1.0
30     rate = (1 + i / n) ** (n / p) - 1
31
32     # pack the reply
33     packInt(RATE_PER_PAYMENT_PERIOD, SIMPL_BIN)
34     packDouble(rate, SIMPL_BIN)
35
36 def annualInterestRate():
37     # unpack the balance of the message
38     n = unpackInt(SIMPL_BIN)
39     p = unpackInt(SIMPL_BIN)
40     r = unpackDouble(SIMPL_BIN)
41
42     # calculate the annual interest rate
43     p *= 1.0
44     annual = n * ((r + 1) ** (p / n) - 1)
45
46     # pack the reply
47     packInt(ANNUAL_INTEREST_RATE, SIMPL_BIN)
48     packDouble(annual, SIMPL_BIN)
49
50 # SIMPL name attach
51 nameAttach("RECEIVER", 1024)
52
53 while 1:
54     # receive incoming SIMPL messages
55     messageSize, senderId = receive()
56
57     # unpack the message token
58     token = unpackInt(SIMPL_BIN)
59
60     if token == PAYMENT_AMOUNT_PER_PERIOD:
61         paymentAmountPerPeriod()
62     elif token == RATE_PER_PAYMENT_PERIOD:
63         ratePerPaymentPeriod()
64     elif token == ANNUAL_INTEREST_RATE:
65         annualInterestRate()
66     else:
67         continue
68
69     # reply to sender
70     reply(senderId)
```

line 3 The SIMPL function calls and packing/unpacking functions are imported

from the csimpl module.

lines 5-7 Variables defining what will be taken as *token* values are defined.

lines 9-20 A function which will handle the case where the incoming message token equals PAYMENT\_AMOUNT\_PER\_PERIOD is defined.

lines 22-34 A function which will handle the case where the incoming message token equals RATE\_PER\_PAYMENT\_PERIOD is defined.

lines 36-48 A function which will handle the case where the incoming message token equals ANNUAL\_INTEREST\_RATE is defined.

line 51 The always required SIMPL name attach. Note that the program's SIMPL name is "RECEIVER" and that the largest allowable message is 1 kbyte.

lines 53-73 An infinite loop wherein messages are received, information unpacked from the incoming message, calculations made, calculated data packed into reply messages, and reply messages returned back to the sending program.

line 55 Messages are received. The messages are contained within a buffer that is defined within the csimpl module.

line 58 The value of the incoming message token is unpacked.

lines 60-67 The value of the token determines the course of action taken.

line 31 The appropriate function is called to take the desired course of action.

lines 9-11 The various message data items are unpacked.

line 14 The amortized payment is calculated for each payment period.

lines 17-18 The reply token and the calculated payment is packed into the reply message.

line 70 The appropriate reply message is sent back to the sending program.

The receiver3.py program is a very straightforward extension of the receiver2.py program. In order to increase the functionality of the receiver program, one merely has to add the appropriate token values, extend the if-else logic to include these new tokens, and add the necessary functionality for the required processing.

## 5.4 Summary

The SIMPL core library transports a contiguous blob of bytes from module A to module B. To become a SIMPL message that blob of bytes needs a structure. There are many possible ways to structure a SIMPL message. Tokenized binary messaging is one possible option.

SIMPL doesn't require tokenized binary messaging to work; it simply works much better with it.

Tokenized binary messaging doesn't require the 'C' language, it is just a natural fit with that language. Tokenized binary messaging can readily be accommodated in Java, Tcl, and Python - the other common languages used to create SIMPL modules.

The biggest advantage of tokenized binary messaging in SIMPL applications is their enhanced extensibility. The disadvantage is that it is a binary formatting scheme which becomes more problematic to implement in SIMPL applications deployed across heterogeneous hardware or wide area networks.

An important point to note is that all of the receiver examples listed above were tested against a *single sender program* written in 'C'. All of the programming languages that support SIMPL do not contain any boundaries between each other with respect to SIMPL communications. That is, any SIMPL program written in any SIMPL-supported language may SIMPL communicate with any other.

## 5.5 Next Chapter

In the next chapter we are going to take a look at the SIMPL public softwareIC repository. This repository forms the seed code for most new SIMPL users' private code repository. The code in this repository uses tokenized messaging extensively so it also forms a great set of examples for what was discussed in this chapter.

# Chapter 6

## SoftwareICs: Practice

### 6.1 Applied SoftwareICs

We have already introduced the concept of a softwareIC and why that concept offers software designers a better way to encapsulate complexity. The SIMPL project maintains a public softwareICs repository on the website. In most cases the softwareIC code in the repository represents framework code from which a developer would customize the application specific softwareIC. The degree to which customization is required depends on the specific softwareIC and ranges from header file tweaks to addition of custom algorithm logic. All the code in the repository is accompanied by working examples of simple implementations complete with full SIMPL testing frameworks. As such this public repository is also a great starting place for new developers to SIMPL to pull seed code from for their own private repositories. In this chapter we will be discussing the softwareICs in the public repository in more detail. You can follow along by first installing the public repository code according to the procedure described in Appendix A.

While all the working examples in the repository are written in ‘C’, there is no reason that they could not be rewritten in Python, Java, or Tcl. As for the softwareICs themselves they are also written in ‘C’. In most cases, while they could be recreated in Python, Java or Tcl, there is little advantage to doing so.

The most basic SIMPL processes types are:

**senders** - those processes who compose messages and wait for replies

**receivers** - those processes who wait for messages and compose replies

All the building blocks discussed in this section will be composites or special types of these two basic building elements.

Over the years, as SIMPL developers, we have adopted a style for SIMPL figures, of which there are several in this chapter. The entire picture represents the SIMPL application. Within that picture the major boxes represent the SIMPL modules associated with the SIMPL application. Within those modules the **S** or **R** denote the ***Send*** or ***Receive*** ports for that module. The direction of the arrowed line represents the major data flow direction for the SIMPL message.

## 6.2 Simulator

One of the great advantages of the SIMPL paradigm is that it allows for the ready development of testing stubs. We have adopted the following naming convention with respect to these testing stub processes.

**stub** - small program that contains the essential aspects of a SIMPL sender or a receiver.

**simulator** - the stub for a receiver process (*sim* for short).

**stimulator** - the stub for a sender process (*stim* for short).

A typical simulator setup might look like Figure 6.1:

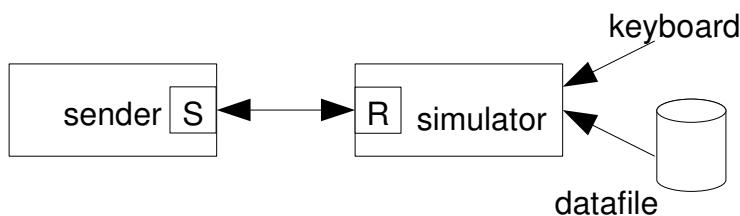


Figure 6.1: Simulator

The item being tested here is the sender. Provided that the simulator conforms to the SIMPL naming convention expected by the sender and conforms to all the expected message formats that the sender can exchange.

**The sender will not be able to detect that it is talking to a test stub.**

Why is this different than employing stubs in non-SIMPL designs? For one thing the stub and the SIMPL module being tested are separate executables. This means that the stub and the SIMPL module could be written in a different language. It may take less developer resources to create a test stub in Python, Java, or Tcl than the equivalent in ‘C’. A ‘C’ based sender cannot detect that it is talking to a Tcl simulator. Since the stub is a separate executable it can be changed and enhanced without the possibility of damaging the integrity of the code under test. In fact the entire sender module can be vigorously tested in a realistic test environment without having to alter the final deployed executable in any fashion. There is no need for conditional compiles, test flags or custom code blocks that are typical of unit test scenarios in non-SIMPL designs. Once tested, the sender executable can be deployed as is, even on another network node.

The exact composition of the simulator code is highly dependent on the application. Figure 6.1 above illustrates a typical scenario whereby the simulator is exercised manually via keyboard commands. In cases where it is impractical to enter canned responses manually those are being fed in from a datafile. We have utilized more sophisticated simulators where the whole test is sequenced, controlled and driven from that data file.

## 6.3 Stimulator

When the object needing unit testing is a receiver one would typically use a stimulator to pretend to be the real sender in the test phase.<sup>1</sup>

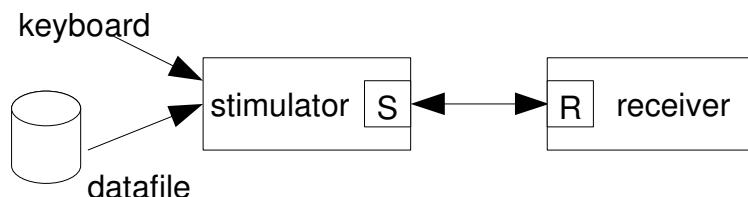


Figure 6.2: Stimulator

The item being tested here is the receiver. As was the case with the simulators above the key here is that provided that the stimulator conforms to all messaging

---

<sup>1</sup>The reader must exercise care as the word **stimulator** and **simulator** differ by a single letter but they represent very different test stubs.

and naming conventions the receiver process will have no way of knowing that it is being sent a message from a stimulator as opposed to the real sender in the final application.

**The receiver will not be able to detect that it is talking to a test stub.**

As was the case with the sender process in the simulator example the receiver under test here can be the final deployable executable in all respects. Once again no conditional compilation or other executable altering techniques are required in the SIMPL paradigm. The stimulator could be written in a language different from the receiver under test. If in the real application the receiver gets messages from multiple senders, a single stimulator could pretend to be all those senders.

As with the simulator, the sample stimulator contains a keyboard interface for the tester to interact with. More sophisticated stimulators may feed the test input from a data file.

## 6.4 Relay

In a SIMPL system all processes are named. A sender has to know the SIMPL name of the receiver in order to open a communication channel with the ***name\_locate*** call. SIMPL contains no built in name sharing mechanism. The application developer has to create a scheme for sharing names between modules which will communicate with each other. In the very simplest of systems these SIMPL names can be hard coded. While this works for sample code, in most SIMPL applications the names are assigned to processes (and passed to other processes) as part of a startup script. The sample code in the SIMPL softwareICs repository uses this name sharing mechanism.

There are times when this manual passing of SIMPL names becomes cumbersome to manage efficiently. In those instances it is better to architect the SIMPL application with a single *well known* portal process to which all senders name locate. ie. the true intended receiver's SIMPL name is *hidden* from the sender.

An example might be an application where a GUI configurator module (receiver) is started and stopped throughout the period that the entire application is active. Another example might be where a customized trace logger (receiver) is dynamically started after the main application has started. Yet another example might be where the application wants to manage alternate logic through

a network outage. In all these cases a relay based portal (Figure 6.3) would be a solution.

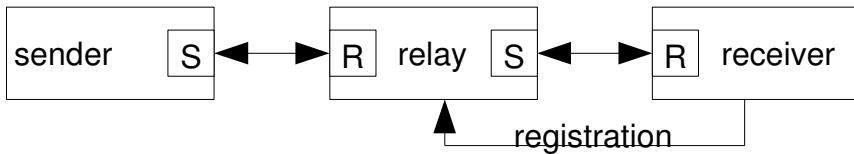


Figure 6.3: Relay

All the SIMPL application logic in Figure 6.3 is contained in the sender and the receiver. Algorithmically they behave exactly as if they were a SIMPL sender-receiver pair and the relay didn't exist.<sup>2</sup> While Figure 6.3 illustrates the relay with a single sender, there is no reason that there couldn't be multiple senders communicating with the single receiver via the relay.

The actual code associated with the sender does not need to change in any way to accommodate the relay. This means that the sender can be unit tested against a receiver and then deployed without change in an application which uses a relay. The sender thinks that the relay process is the intended receiver for its messages. It does all the normal name locate and send operations with the relay as if it were the final processor of the message. As such the sender needs to know the SIMPL name of the relay but does not need to know the SIMPL name of the receiver. The sample code in the repository contains a 'C' based sender. Python or Tcl/Tk could have been used to create the sender which in turn interfaced with the 'C' based relay softwareIC.

The receiver on the other hand, needs to inform the relay that it exists in order for the relay to store and forward messages to it. As such the receiver needs to know the SIMPL name of the relay. To perform this *inform the relay* step the relay construct utilizes a special tokenized SIMPL message with the REGISTRATION token. As part of the initialization sequence the receiver needs to become a temporary sender, open a communication channel to the relay and send it a REGISTRATION message broadcasting its SIMPL name. Once this registration is accomplished the receiver resumes its normal receive loop and responds to incoming messages in the normal manner. Hence the receiver code

---

<sup>2</sup>There is relay example code in the SIMPL SoftwareICs repository which can be installed according to the procedure in Appendix A

does need to be changed to accommodate the relay. However, those changes are relegated to the initialization section of the code and not the ***Receive-Reply*** algorithm section. As with the sender, the example code for the receiver in the repository is ‘C’ based. It could have been written in Python or Tcl/Tk.

Upon receiving the REGISTRATION message the relay opens a communication channel to the receiver by doing a name locate with the supplied name so that it can store and forward any messages. Because the relay intercepts and handles the REGISTRATION message internally the namespace associated with the application’s messaging protocol needs to be adjusted accordingly. The sample relay code in the repository uses a basic 16 bit word as the token in a tokenized message passing protocol. To accommodate the possibility of a registration and error token the namespace is expanded to include two new tokens: RELAY\_REGISTER and RELAY\_ERROR. In the repository example the sender and receiver exchange a TEST tokenized message. When the sender issues a TEST message the relay process simply copies it through to the registered receiver process and the application behaves as if the relay didn’t exist. REGISTER and ERROR tokens are intercepted and handled by the relay construct itself.

The relay will incur a performance penalty over the straight sender-receiver message exchange but the advantages which come with the construct often outweigh the downside. The relay is a powerful SIMPL construct.

## 6.5 Proxy

The proxy, like the relay, provides portal like SIMPL receiver name *hiding*. The proxy differs from the relay in how the receiver *registers* and in the fact that the proxy allows for multiple instances of a receiver to be prestarted.

An example where a proxy would be used is in a high transaction rate application such as a credit card authorization system. In such an application the credit card terminals would represent multiple SIMPL senders all requiring the services of a centralized processing house (SIMPL receiver). Using SIMPL one could architect such a system without a proxy. This would allow the SIMPL receive FIFO to naturally queue all the credit card authorization requests. While this would work, the overall performance would be governed by the single receiver’s throughput. At times when the rate of credit card transactions is high, credit card users would experience authorization delays as their request sits in the queue waiting to be serviced. One way to speed things up would be to re-engineer our application to dispatch each sender’s request to a separate instance

of our receiver. This would minimize the waiting in the queue. However, since creating a separate instance of the receiver itself takes time and resources it would be helpful to prestart a number of receiver instances to buffer in the event of a burst in authorization requests.

The proxy software IC illustrated in Figure 6.4 would allow us to build such an application.

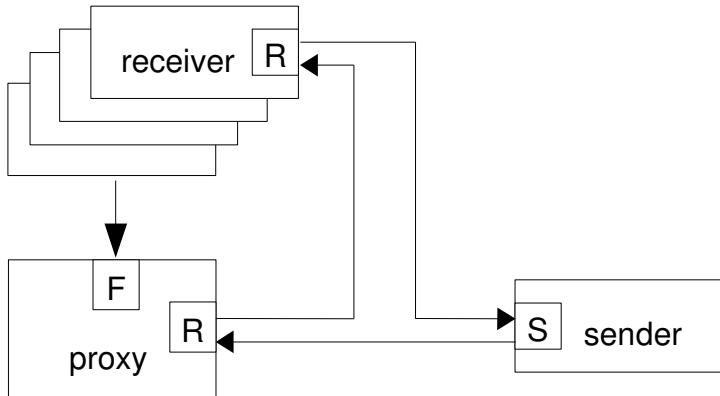


Figure 6.4: Proxy

The proxy IC makes use of intimate knowledge of the workings of SIMPL FIFOs to achieve a transparent relaying of messages from a sender to a one of the prestarted receivers.

To utilize the proxy, the sender code needs no modification over the straight sender-receiver pairing. The sender believes that the proxy is the receiver processing the messages. As such the sender only needs to know the SIMPL name of the proxy. Algorithmically the sender behaves exactly as if the message were exchanged directly with the receiver.

The receiver code must be adjusted to become proxy *aware*. As was the case in the previous relay example, the receiver needs to make itself known to the proxy. The relay used a SIMPL message for this purpose. The proxy uses a special purpose FIFO (box F in Figure 6.4) for this communication. The special purpose FIFO has two advantages over the SIMPL message:

- There is no need to *interfere* with the token namespace to accommodate proxy specific tasks like registration.
- The FIFO naturally queues prestarted instances of the receiver.

The relay used a single registration during the receiver initialization. The proxy typically re-registers each time through the transaction loop as shown by the position of the ***postmyID*** call in the code snip below:<sup>3</sup>

```

1 token=(UINT16 *)inArea ;
2
3 nbytes = Receive(&sender , inArea , MAX_MSG_SIZE) ;
4
5 switch(*token)
6 {
7     case RELAY_TEST:
8         {
9             RELAY_TEST_MSG *inMsg ;
10            RELAY_TEST_MSG *outMsg ;
11
12            inMsg=(RELAY_TEST_MSG *)inArea ;
13
14            outMsg=(RELAY_TEST_MSG *)outArea ;
15            outMsg->token=RELAY_TEST ;
16            sprintf(outMsg->str , " reply %d" ,msgCount++);
17
18            Reply ( sender ,outArea , sizeof(RELAY_TEST_MSG )) ;
19
20            postmyID ( myID ) ;
21        }
22        break ;

```

where the ***postmyID*** function is essentially a ***write*** to the special FIFO as illustrated below:

```

1 int postmyID ( theID )
2 {
3     int rc=-1;
4
5     rc=write ( proxyID , &theID , 4 );
6
7     return ( rc );
8 } // end postmyID

```

There is another difference between the relay and the proxy. The relay is intimately involved in handing the SIMPL message and the SIMPL response in a store and forward manner. The proxy on the other hand, utilizes the SIMPL API ***Relay*** command to reroute the incoming message directly to the

---

<sup>3</sup>There is a full proxy example in the softwareICs repository which can be installed using the procedure in Appendix A

intended receiver without actually doing a store and forward. As such the proxy is more efficient and is not involved in the reply portion of the message exchange. This proxy mechanism has several advantages, but it means that the proxy is restricted to having its prestarted receivers on the same network node as the proxy itself resides. For most applications where the proxy would be deployed this is not a severe restriction.

## 6.6 Stack

Many messaging protocols can be visualized as layers to help facilitate modularity in the processing. The OSI model for network protocol design is an example. One construct which lends itself well to this abstraction is called a stack.

An example where a stack might be used is in a web services application such as video surveillance system. In such an application a CGI module (SIMPL sender) would be spawned by the web server and would compose and emit a message. That message may go through several translation layers before arriving at a SIMPL receiver who is responsible for setting the camera parameter. Each layer in the stack would add value to the original message. The last layer in the stack would consume the resulting message and respond to the original sender.

The stack softwareIC is illustrated in Figure 6.5.

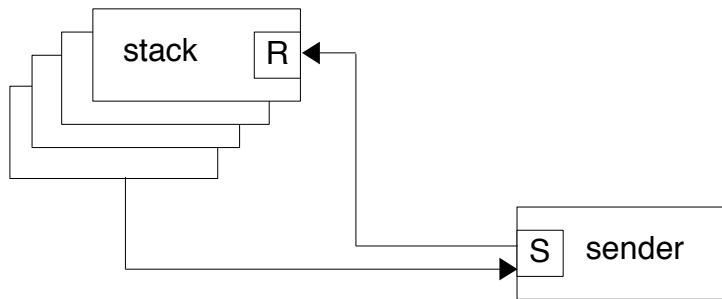


Figure 6.5: Stack

The stack IC makes use of the Receive - Relay construct in its upper layers to hold the sender blocked throughout the transaction. More efficient stack algorithms make use of the special form of the Receive call whereby the message

is left in and manipulated directly in the shared memory buffer. The last process in the stack would do the Reply to unblock the message originator. The sender sees the entire stack as if it were a straight SIMPL receiver. Most often the entire stack resides on a single network node, but this not a requirement imposed by SIMPL. Since each layer in the stack is SIMPL process there is no technical reason that they couldn't be distributed across several network nodes.

To utilize the stack, the sender code needs no special knowledge. It simply composes and sends a message as if it were communicating with a straight sender-receiver pairing. The sender believes that the lead process in the stack is the receiver processing the messages. As such that is the only SIMPL name the sender needs to know.

The code in each layer of the stack must, by definition, be aware of its position in the stack. Each layer is responsible for a particular aspect of the composition of the overall transaction message. That message must flow through the various layers in the correct order. If the message is modified insitu in the shared memory the code is only marginally different from that associated with a straight SIMPL receiver as shown in the code snip below:<sup>4</sup>

```

1 nbytes = Receive(&sender , NULL, 0 );
2 sArea=whatsMyRecvPtr( sender );
3
4
5 token=(UINT16 *)sArea ;
6
7 switch(*token)
8 {
9     case STACK_TEST:
10    {
11        STACK_TEST_MSG *inMsg ;
12        STACK_TEST2_MSG *outMsg ;
13
14        inMsg=(STACK_TEST_MSG *)sArea ;
15
16        // process incoming message here
17        outMsg=(STACK_TEST2_MSG *)sArea ;
18
19        // load outgoing message here
20        Relay(sender ,relayTo .channelID );
21    }
22    break;

```

---

<sup>4</sup>There is a full stack example in the softwareICs repository which can be installed using the procedure in Appendix A

Each of the stack layers is a separate SIMPL process and because context switching must occur as the message flows through each layer, there will be a performance penalty over combining all the logic into a single receiver. The advantages of modular design, usually outweigh such considerations in SIMPL systems where the stack IC would be employed.

## 6.7 Courier

Occasionally it is necessary in a design for two SIMPL receiver processes to exchange messages. A typical example would involve a user interface (UI) process. User interface processes, be they simple text based screens or GUI's, are event driven. As such they are natural SIMPL receiver type processes. They can handle incoming messages easily. However, most user interfaces dispatch outgoing requests as well. The most straightforward way to dispatch a message with SIMPL is to send it. However the ***Send*** in SIMPL is a blocking call. If you went ahead and coded a blocking ***Send*** into the UI and the receiver was busy, the interface could appear to *freeze* while the request was awaiting service. This is not the desired behaviour.

How then does a user interface (UI) get information from another SIMPL receiver? The courier construct illustrated in Figure 6.6 is a good way to accommodate this requirement.

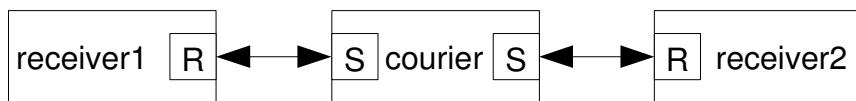


Figure 6.6: Courier

Let us assume that in Figure 6.6 receiver1 represents the GUI and receiver2 represents the SIMPL process that the user wants information from. ie. receiver1 initiates the SIMPL message and receiver2 processes it.

Receiver2 needs no modifications to accommodate the courier.

Receiver1 (the UI in our example) does need some modifications to be made courier aware. The first modification involves maneuvering the courier into a reply blocked state. When the courier process is started the first thing it does

is locate the UI process it is designated to service. Once located, the courier will send a registration type message to that process indicating that it is ready for action. The UI process will simply note that the courier is available and not reply, thereby leaving the courier reply blocked.

The second modification involves dispatching an outgoing request to the receiver2 process by composing and replying a message to that blocked courier. Because **Reply** is a non-blocking call, there is no chance that this will result in the UI *freezing* as might be the case if a blocking call was used. The courier is now free to forward the message to the receiver2 process using a blocking **Send**. At this point the courier is reply blocked on receiver2 and the UI is completely free to do other things as permitted by its logic.

When receiver2 finishes processing the message it replies to the courier. The courier simply forwards that reply on to the UI process using a blocking **Send** and once again becomes reply blocked on the UI. ie. returns to its original state. The UI receives this message in the normal manner, notes that it came via the courier, marks that the courier is once again available and processes the message in accordance with the logic coded.

The simple courier described above is the variation stored in the SIMPL softwareICs repository. It is a single request version. If a second UI request intended for the receiver2 process is generated within the UI before the courier returns its first response, that request will be refused citing the *busy courier*. A simple enhancement to this single request logic is to have a single message queuing capability in the UI. The *busy courier* response then would only come if a third UI request is attempted before the original response is received. In most UI processes this single message queue is more than adequate. A larger queue depth algorithm could be constructed readily, but the need for this is often indicative of a poor UI design elsewhere.

Another variation on the courier model is to have a parent process fork the couriers on demand. In some cases this capability is more desirable than having the courier prestarted along with the GUI process. The web applet type GUI applications are examples where this courier spawning technique is desirable. Especially in user interface designs, the courier construct is a very useful SIMPL building block indeed.

## 6.8 Agency

One of the advantages of SIMPL is that there is very little algorithmic difference between a receiver's **Send** driven message loop and a sender's **Reply** driven

message loop. Behaviourally however, there is a fundamental difference. In the first case the receiver holds the sender blocked until the message is processed. In the second case the message initiator (the receiver) is not held while the message is processed.

In SIMPL application designs where the sender does the message processing, it is often necessary to exchange messages between two SIMPL senders. For this the agency construct illustrated in its simplest form in Figure 6.7 is used.



Figure 6.7: Agency

In Figure 6.7 the requestor is simply another name for the normal message initiator: the sender. Similarly, the agent (also a SIMPL sender) contains the logic for processing the requestor's SIMPL message, logic that normally would exist in a SIMPL receiver. Variations to this basic agency framework all have to do with the blocking logic applied to the requestor message. These variations can span the full spectrum from full blocking (as in our example) to immediate unblocking with message queuing (as in the broadcaster softwareIC example below).

The agency construct is a natural SIMPL message router. Often times different portions of the tokenized message namespace are allocated to separate SIMPL agent processes. An example of this might be a message processing stack. Each layer in the stack is handled by a separate agent.

The agency construct is also a natural gateway. Sometimes an application specification calls for the gateway to assert control on the message flow and sequencing to the agent processes. Straight SIMPL messaging is always immediate delivery with a democratic first in first out queue if the receiver is busy. Agency based gateways can introduce queuing and priority based message delivery algorithms. This agency framework forms the basis for several other softwareICs in the SIMPL repository:

- Broadcaster
- Scheduler

- Statemachine

The softwareICs we have discussed so far (stimulator, simulator, relay, proxy and courier) all share the distinction that stock SIMPL senders and receivers can be adapted to work with these ICs in a straightforward manner, sometimes without any changes. The requestor, agent and agency form a much more connected package. While stock sender and receiver algorithmic logic can readily be adapted to work as requestors and agents, the framework changes are more fundamental. An agent cannot stand as a SIMPL process on its own. It is tightly coupled to its agency.

As such, the agency framework makes use of what are termed wrapped SIMPL messages. A wrapped SIMPL message is a tokenized SIMPL message embedded inside the data area of another tokenized SIMPL message. In our example the outer layer message is used by the agency. The inner layer message is produced by the requestor and consumed by the agent. The requestor and the agent build the two layer messages to route their cargo through the agency.

Other than the need for SIMPL wrapper messages, the requestor and agent behave exactly as if the requestor were sending its message to a basic receiver. In fact there is very little difference in the requestor code for dealing with agencies. Why then go to all this trouble?

First of all, it is now possible to dynamically start and stop the agent process in this system without affecting the requestor. In systems where the agent is undergoing significant revisions or upgrades this is a distinct advantage.

Secondly, the requestor in this system does not need to know the name of the agent in order to exchange a message with it. Like the relay and proxy, the agency construct can be viewed as a message gateway.

To understand the further advantages, we need to examine the case where we may have multiple requestors all talking to the same agency and agent. In this scenario the agency will actually receive and queue the requestor's messages. The agency logic can then be in control of the order in which these messages are dispatched to the agent. In a normal sender/receiver pairing the FIFO imposes a strict first in first out ordering and it is not possible to have a higher priority message jump ahead in the queue. In the agency construct this is very possible.

In addition, in the normal SIMPL sender/receiver pairing the messaging is synchronous. It is intentionally difficult to kick a sender out of a reply blocked state other than by having the receiver do a reply. This means things like timeouts or *aged data* are difficult to handle. The agency construct makes these things relatively easy to manage. While messages are pending in the agency queue the agency can be kicked into examining these periodically for timeouts

or aging.

Finally, in normal SIMPL message exchanges there is no concept of delayed, repeated or scheduled delivery. Once again the agency construct makes all these forms of message delivery possible.

The agency construct will suffer a performance penalty when compared against a basic sender/receiver pair because at least two extra messages need to be exchanged in each transaction. The agency framework however, is a powerful one and has been used in several of our projects over the years.

## 6.9 Broadcaster

There are times in a design where there is a need for a one to many sender/receiver relationship. An example might be a SIMPL application with multiple GUI interfaces that all need to be fed the same data. Another example might be an application which needs an audit logger for recording all message traffic to a GUI screen. For simple cases one can simply have the sender locate all the intended recipients and loop through sending to each. In more sophisticated designs the broadcaster construct illustrated in Figure 6.8 is used.

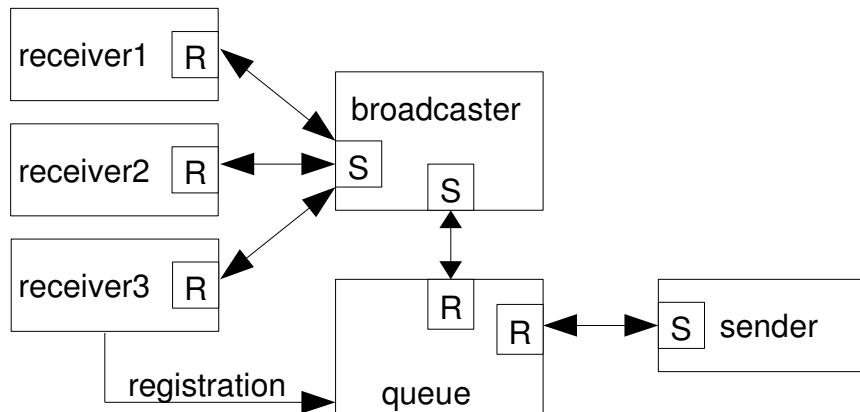


Figure 6.8: Broadcaster

The actual application code is contained in the sender and receivers in Figure 6.8. Since this is a one to many message relationship, the actual message protocol between the sender and any receiver has to include a simple NULL reply since SIMPL has no mechanism to push multiple replies in response to a single ***Send***.

In cases where the broadcaster would be used this is not a severe messaging protocol restriction. During normal operation any message transmitted by the sender will get distributed to all the registered receivers (eg. receiver1,2,3 in Figure 6.8).

The broadcaster-agent and broadcaster-queue are simplified derivatives of the agent and agency construct we discussed earlier. The broadcaster-queue stores and forwards all message traffic it receives to a single broadcaster-agent process. The broadcaster-queue generates the NULL reply to all incoming messages. The broadcaster-agent maintains a list of SIMPL receivers to send to. The broadcaster-queue utilizes the same token namespace shifting technique employed by the relay to accommodate the broadcaster specific tokens: REGISTER, DREGISTER and WHAT\_YA\_GOT. This technique supplants the more sophisticated SIMPL message wrapping technique employed by the more general purpose agency-agent construct.

As such the sender need not be modified in any way to work with the broadcaster. The sender simply treats the broadcaster-queue as the intended receiver of its messages. Since the expected protocol includes a NULL reply the sender is unable to distinguish that it is dealing with a broadcaster-queue and not the actual intended receiver(s). In fact the sender can be unit tested against those intended receivers and then simply deployed via the broadcaster.

The receivers on the other hand, need to be broadcaster aware in much the same way that receivers connected to a relay were relay aware. The broadcaster needs to know the SIMPL names of the receivers it is broadcasting to. This information is communicated via a special tokenized message using the REGISTER token during the initialization sequence for the receiver. Similarly the receiver issues the DREGISTER token to the broadcaster at any time it no longer wishes to receive the transmissions. The broadcaster will automatically deregister any recipient upon a transmission failure. Outside of the registration and deregistration sequences these receivers process messages as if they came directly from the sender.

The WHAT\_YA\_GOT token is utilized internally between the broadcaster-agent and the broadcaster-queue.

A typical sequence may start as follows. A receiver (say receiver1) decides that it wishes to receive broadcast messages. As part of that sequence it sends a registration type message to the broadcaster-queue process. This inserts this receiver into the broadcaster-agents' broadcast list. At this point the sender may send a message to the broadcaster-queue process. The broadcaster-queue will immediately acknowledge this message with a NULL reply. Each receiver in the broadcaster-agents' list will receive a copy of this message. All receiver replies

will be absorbed by the broadcaster-agent. If, while the actual broadcasting is taking place, a sender transmits a new message to the broadcaster-queue that message is simply placed on the queue for later transmission.

While Figure 6.8 illustrates a broadcaster with a single sender, there is no restriction on the number of senders that can transmit messages to the broadcast-queue. For cases like the audit logger or synchronization of multiple GUI screens with the same data stream, the broadcaster construct is a powerful SIMPL softwareIC.

## 6.10 Scheduler

All SIMPL messages are delivered immediately subject to the democracy of the first in first out (FIFO) queuing that occurs if the receiver is busy. There are times in a design where there is a need to schedule messages for delivery at some future time. In addition, one might want that message to be scheduled regularly on a daily or weekly schedule.

Figure 6.9 illustrates a derivative of the agency which can permit scheduled delivery of SIMPL messages.

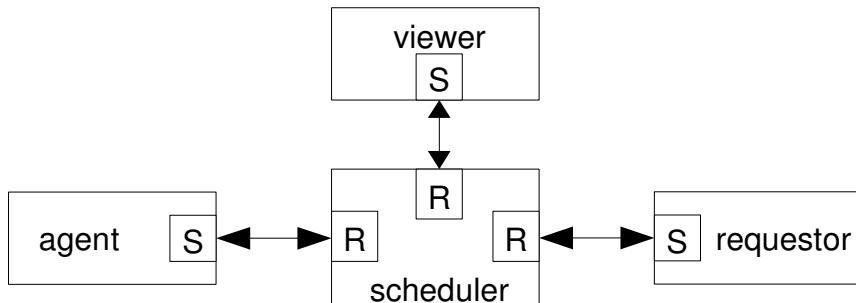


Figure 6.9: Scheduler

The requestor is the message initiator. However, since this message is going to be delivered at some future time, the messaging protocol would likely entail an immediate NULL reply to unblock the requestor. Figure 6.9 shows two agent processes: **agent** and **viewer**.

The example code contains a viewer which is a console viewer of the messages currently queued in the scheduler. Although the sample code illustrates a console process it could very well have been a SIMPL enabled GUI. The agent is the

module which processes the requestors' delayed message. This agent is reply driven from the scheduler queue.

All messages in the scheduler example are tokenized. However, the requestor submits a wrapped SIMPL message. The outer portion is stripped away and used by the scheduler itself. The inner message is passed through to the agent for processing. For the sample code the contents of the outer portion are described by the **SCHEUDLE\_THIS\_MSG** structure in the **schedulerMsgs.h** header. It contains various fields for controlling the scheduling of the inner message.

The main looping sequence in the scheduler is kicked off by the on board timer. Each click of that timer spawns the following activities:

- A check on the queued messages to mark any that have now expired.
- If the agent is available the first available expired message is dequeued and replied to the agent.
- If the dequeued message is on a daily or weekly repeat it is stamped with the next timestamp and left on the queue.

The message to be delivered to the agent is treated as a package of bytes by the scheduler, to be queued until the time comes to forward it on the agent. The agent in this scheme only sees this package of bytes with all the scheduling info stripped away before it is forwarded. Obviously, in the interest of simplicity this example scheduler lacks some features, that could readily be added such as:

- Text (or XML) file driven input of the scheduled message.
- Some form of persistent storage of the schedule queue in the event of a restart.
- Management functionality in addition to viewing the queue such as selecting and deleting messages.

Nevertheless, the scheduler framework in the repository is a very usable softwareIC.

## 6.11 Statemachine

Another derivative of the agency in the repository is the statemachine softwareIC. State machine logic is a very common element of many software applications. State machines are by definition very customized to the problem they

are trying to represent. As such it is very difficult to build a general purpose state machine that works for many classes of problems. Instead this softwareIC takes the approach of a source code framework.

The **SM\_common** directory contains the basic state machine infrastructure and a definition of an API to that infrastructure.

The **SM\_door** subdirectory contains the specific state machine logic and implementations of the state machine API for a very simple four state door (Figure 6.10).

These two source code sections are re-merged by the Makefile into a single executable. The idea is that for another type of system the **SM\_common** stays and the **SM\_whatever** is created which results in a new type of executable. In this manner the **SM\_common** code can be shared across several different executables.

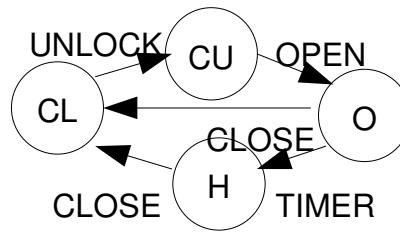


Figure 6.10: Door

Where the states are:

**CL** - closed and locked

**CU** - closed and unlocked

**O** - opened

**H** - held open

The events in this simple door system are:

**UNLOCK** - key is turned to unlock the door

**OPEN** - door is pulled open

**CLOSE** - door is shut (and automatically relocked)

## TIMER - held open timer expires

The sample door illustrated in Figure 6.10 operates in the following manner. With the door in a closed locked state (CL) a resident steps up and inserts a key to unlock the door. This causes our door to transition to the closed unlocked (CU) state. If the resident pulls open the door, breaking a contact sensor, the door transitions to an open (O) state. At this point a timer is engaged to monitor the length of time the door is left open. If the resident allows the door to close the door state transitions back to closed locked (CL) and the timer is cancelled. If the timer expires first the door transitions to a held (H) state and an alarm message is triggered. When the door is finally closed the alarm is cancelled and the door returns to a closed locked (CL) state.

The sample code in the repository is represented in Figure 6.11 below:

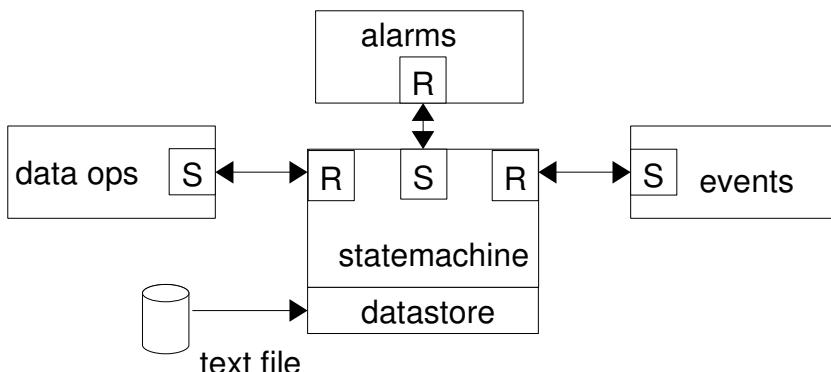


Figure 6.11: Statemachine

The datastore in the sample code is a doubly linked in memory list, but it could easily be substituted by any number of more sophisticated datastores as the application requires. The datastore itself is fed from a simple **tag, value** paired text file. To make this datastore more embedded friendly we have employed the concept of a single block of dynamically allocated memory which is then subdivided into memory pools. The datastore is organized with a separate record for each door. The complete state of a given door is stored in this record so the **statemachine** logic can accommodate a number of doors in a single SIMPL module.

The main **statemachine** is for the most part a SIMPL receiver, but occasionally a SIMPL sender. On the receive port this **statemachine** can accommodate two classes of messages:

- Those associated with operating on the datastore.
  - **ADD**
  - **DELETE**
  - **OVERWRITE**
  - **RESYNC**
- The events themselves.

In the sample code a single test stub called **eventStim** is involved in issuing all these messages via a simple command line interface.

The **eventStim** is also configured to receive any **ALARM** messages that the **statemachine** issues, which in the case of our simple door is associated with the door being held open beyond a specified time window.

## 6.12 Polling Emitter

SIMPL is often used in low speed (less than 100 samples per second) data acquisition/control applications such as building automation. This by definition means that in such an application at least one SIMPL module will be interfacing with hardware to gather and transmit data. If that hardware interface is a serial connection, one class of data acquisition/control application which still finds usage involves polling the serial interfaced hardware. In such cases the SIMPL interfacing module must reconcile the need to poll the serial port, accept occasional data for transmission on that serial port and emit the occasional SIMPL message containing acquired data. The first and last requirement make this process a natural SIMPL sender. The second requirement wants this process to be an occasional receiver.

The polling emitter illustrated in Figure 6.12 tries to square away these conflicting requirements associated with this class of application. This softwareIC solves this problem by making the Polling Emitter an occasional sender while allowing for a small piece of shared memory be used to intercept the polling loop and trigger a *call home* response from the emitter.

In the 6.12 we are showing the emitter connected to a queuing receiver process. While this is strictly not necessary one will want to take care that all messages sent from the polling emitter are blocked for the minimum amount of time.

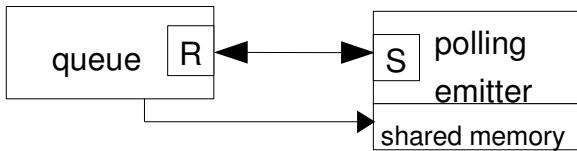


Figure 6.12: Polling Emitter

A typical sequence may start as follows. The queue and the emitter are started up. The emitter name locates the queue and sends an initialization message indicating to the queue where the shared memory block is. The queue process will reply back to unblock the emitter and establish a connection to the emitter's shared memory.

While this shared memory could contain any structured data which the queue and the emitter agree upon. eg. a table of serial devices which are to be polled. We are going to restrict this example to the simplest of configurations where the shared memory contains a single *call home* flag. The queue process will be the changer of this flag and the emitter process will only read it.

Inside the emitter there is a loop which is endlessly cycling around all the hardware it is supposed to be polling. The sample code in the repository uses a one second interval checking on a text file to simulate a polling loop. If a change has been made to that text file, the sample code will reread the file and send its contents on to the queue.

Each pass through that loop we are going to have the emitter check on the flag located in the shared memory area. If that flag is set we are going to interrupt the polling and send a CALL\_HOME tokenized message to the queue process. This gives the queue process an opportunity to stuff something into the reply and then clear the flag.

In this manner we can demonstrate the polling emitter looping around checking on the simulated hardware (a file) and emitting any changes that it observes in that simulated hardware. We can interrupt the polling loop at any time and have the emitter call home to receive some data back from the queue. Presumably this would be a message destined for the hardware that the emitter is polling.

## 6.13 Summary

In this chapter we have presented the SIMPL public softwareICs repository. This is an ongoing effort and changes and additions are made all the time. Most new SIMPL developers find the code in this public repository forms great seed code for their projects.

## 6.14 Next Chapter

In the next chapter we are going to examine how the SIMPL core library can be made network aware without having to change any code. In SIMPL terminology we refer to this type of SIMPL application as being network agnostic.



# Chapter 7

## Surrogates and Networking

In chapter 4 we looked at how SIMPL operates on a single host at a local level. It was mentioned that in order to operate over a network SIMPL utilizes the services of programs called *surrogates*. In this chapter we will examine the surrogate programs and their operation in greater detail as well as some issues pertaining to network communications.

### 7.1 Background

The overriding desire to make network transport of SIMPL messages invisible is the driving force behind the software design of networking SIMPL. On a local level, SIMPL is composed of sending and receiving programs that communicate with each other directly by passing well-defined messages in areas of shared memory. Shared memory is not normally network accessible. This should not prevent SIMPL from transparently transporting messages between nodes.

In order to make network SIMPL function in the same fashion as local SIMPL we have to address the following issues:

1. We need a scheme to replicate and synchronize data in shared memory on one node with shared memory on another node.
2. There can be any number of disparate ways (protocols) that govern the remote communications between hosts that SIMPL might use. Certainly we can think of a few immediately, but what about the future? We need a design that is amenable to extending SIMPL in concert with new and different protocols. This issue is covered below in the section called *Surrogates*.

3. We need a transparent way of name locating a remote program. That is, we want to simply pass in a name string to the ***name\_locate*** call in the same way that we do locally. Recall that name locating is SIMPL's way of making a communication path between sender and receiver. This makes the API identical and accordingly there would be no obvious difference between local and remote SIMPL connectivity as far as the programmer using SIMPL is concerned. Moreover, if the name string passed in to the ***name\_locate*** function is a variable, then it can be anything. The program making the ***name\_locate*** call is then indifferent as to whether the looked for program is local or remote - which is as it should be. How this issue is dealt with is described below in detail in the section called *Remote Name Location*.
4. Given that SIMPL might use different protocols in order to communicate remotely we will need some way to decide which method of communication is desired and/or available. This issue is dealt with below in the section called *Protocol Router*.

The blocking messaging protocol that local SIMPL uses makes a transparent networked SIMPL possible.

## 7.2 Surrogates

Surrogates are aptly named. They are programs that invisibly replace actual senders and receivers so that SIMPL network communications appear to be local and seamless. Surrogates come in two varieties, namely senders and receivers, depending on which type of SIMPL module they are standing in for.

Suppose that a sender on one host needs to send a message to a receiver on another remote computer. To maintain a seamless API, this sender program will make a ***name\_locate*** call seeking a connection to the remote receiver. Conceptually, if we can *arrange* for this sender to *think* that a receiver surrogate of the same name exists locally, this local SIMPL connection can be made to succeed. Similarly if we can *arrange* for the receiver to *think* that a sender surrogate is to be the origin of any messages, another local SIMPL connection can be made to succeed. Finally, if we can arrange for the surrogate pair to agree on a network protocol for inter-surrogate communication, we can use local SIMPL functions to pass messages over the resulting composite channel. What actually happens is a bit more complicated.

Firstly, the sender program, via the ***name\_locate*** call will ascertain if such a remote receiver exists and can be reached. If so, a surrogate receiver will be started on the computer that is running the sender program. At the same time, a surrogate sender will be started on the remote computer that is home to the remote receiver program. These two surrogate partners (surrogate sender and surrogate receiver) agree on a network protocol to be used for communication. The SIMPL name of the remote receiver is then communicated across the network to the surrogate partner (surrogate sender). Armed with this name the surrogate sender can do a local ***name\_locate*** to complete the full communications channel from local sender to remote receiver. In this way the local sender's ***name\_locate*** can now return a valid SIMPL local channel ID: that of the receiver surrogate. The sender can then use local SIMPL functions to send its message. The recipient of that message is a surrogate for the remote receiver. While the sender remains blocked awaiting a response, the receiver surrogate can safely transmit the contents of the message using the network protocol to its surrogate partner (the surrogate sender) on the remote node. When the surrogate sender receives the message on the network channel it simply uses a local ***Send*** to deliver the message to the intended receiver. The response travels back in exactly the reverse manner. As far as the original sender and receiver programs are concerned, they are communicating directly with each other. See Figure 7.1

The surrogate receiver/sender pairs communicate with each other via some sort of network communications protocol. At the time of this writing there were two such computer communications protocols in use. The first, and the most important of these is based on TCP/IP sockets. TCP/IP is the protocol for the Internet and is available on most hardware platforms as a result. As long as a host supports a TCP/IP stack and is connected in some fashion to another host with a TCP/IP stack, then SIMPL communication can occur (within security limits of course). This TCP/IP surrogate type is useful on local and wide area networks as well as the Internet.

The second method of communication is more restrictive. It employs a RS-232 communications protocol. In many wired RS-232 situations the throughput is less than Ethernet enabled TCP/IP. Similarly the effective range for RS-232 can be relatively short. However, with the advent of RS-232 radio modems this distance can be expanded to 25km wirelessly. The RS-232 surrogate protocol still has its uses. For one it can act as a failover backup for a primary TCP/IP network. For some classes of devices the restrictions associated with RS-232 are not an issue and this is an acceptable protocol for SIMPL communication. Since RS-232 is typically a point to point protocol it can be more secure than a routed TCP/IP protocol.

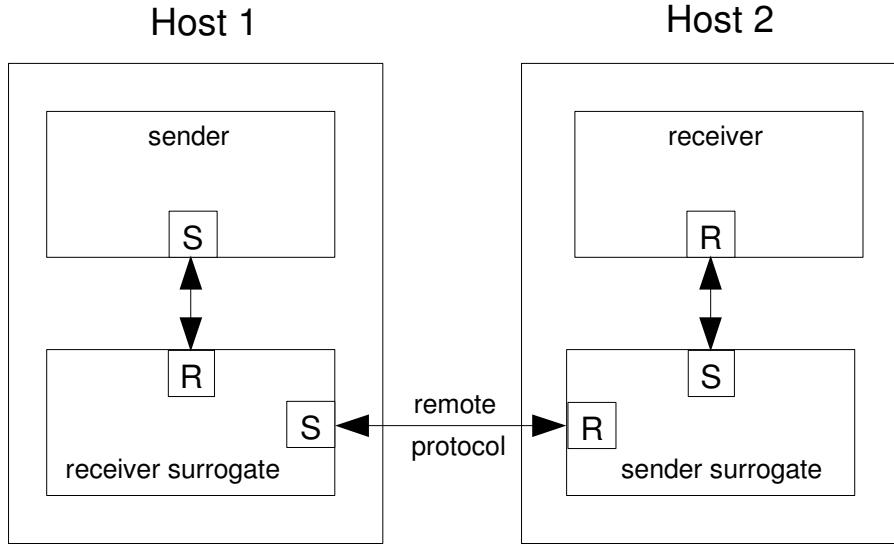


Figure 7.1: Surrogate Programs

These two surrogates follow roughly the same general framework. If one wanted to make surrogates that used a RS-422 protocol for example, it would be very straightforward to modify the current RS-232 mechanism. In some of the following sections we will explore more of the details of the TCP/IP and RS-232 approaches.

Like the core SIMPL library, the SIMPL surrogates are written in ‘C’ language. The surrogates themselves link to the SIMPL core library in order to function. As such we do not consider the SIMPL surrogates to be core to SIMPL. In a very real sense surrogates are merely specialized SIMPL modules. Having said this the surrogates are included in the main SIMPL package because they provide transparent networking which greatly increases SIMPL’s functional applicability.

### 7.3 Remote Name Location

From a programmer’s standpoint, the connection between a strictly local use of SIMPL (restricted to communications on one host) and remote SIMPL is the ***name\_locate*** call. The mechanics of this function call are described in detail in Appendix C. The syntax of the ***name\_locate*** call is unchanged between a local and a remote SIMPL call. There is still only a single name string as

the functional argument. The call still returns the SIMPL ID of the local process pointed to by the SIMPL name. What varies between a local and remote call is the format of the name string. The name string that is passed into the **name\_locate** function call takes the form:

**protocol name:host name:process name**

Note the demarcation of the three parts of the name string by colons. Only the process name portion is compulsory. The other two parts are optional. The *host name* must be specified if the sending and receiving processes are running on different host computers. The *protocol name* must be specified if there are more than one type of surrogate protocol available and the desired protocol is not running as the default.

Now, let's look at each of these values in turn. The *process name* determines the SIMPL name of the receiver that the sender wants to communicate with. This is all that has to be filled in for local SIMPL communications. Suppose that we have a sender with the SIMPL name 'Newton' that wants to send a message to a local program with the SIMPL name 'Einstein'. The Newton program would simply make the following call:

**id = name\_locate("Einstein");**

Suppose now that both Newton and Einstein are running on a host computer called 'Earth'. Another variation that will also work for local name locating would be the following call:

**id = name\_locate("Earth:Einstein");**

Since both Newton and Einstein are running on Earth, then the **name\_locate** call would also be treated as local SIMPL communications. No surrogates will be started and local SIMPL will be used to transmit the message.

Suppose now that the Newton sender program is running on a host computer called Earth but the Einstein receiver program is running on a host called Mars. In this case, Newton would make a **name\_locate** call as follows:

**id = name\_locate("Mars:Einstein");**

If this call is successful a surrogate pair will be set up between Newton and Einstein. That is, Newton and a surrogate\_r will be *local SIMPL connected* on Earth, Einstein and a surrogate\_s will be *local SIMPL connected* on Mars and the surrogate\_r and surrogate\_s will be *remote SIMPL connected* via whatever protocol and hardware is the default.

If there is only one type of surrogate daemon running on the Earth host then the above **name\_locate** call will be adequate. If there is more than one type of surrogate protocol available between Earth and Mars, it may be preferable to

choose the surrogate type directly rather than leave it up whatever is available at the time of the ***name\_locate*** call. Leaving the protocol name value vacant is a signal that the default surrogate will do. We may choose to explicitly direct the protocol to be TCP/IP. In this case, the Newton program will make the call as follows:

```
id = name_locate("SIMPL-TCP:Mars:Einstein");
```

This would force the surrogates to be of type TCP/IP.

## 7.4 Protocol Router

The *protocol router* is a program that must always be present if remote ***name\_locate*** calls are made; the binary file name for this program is called protocolRouter and is generally run as a background process. The purpose of this program is to keep track of the latest available surrogate\_r programs per protocol. When surrogate daemons (surrogateTcp and/or surrogateRS232) are started, they fork into two quite separate programs, viz. surrogate\_R and surrogate\_S. The uppercase R and S signify that these two programs will be the parents of the various surrogate\_r and surrogate\_s programs respectively. The various surrogate\_r and surrogate\_s are the actual programs that provide the remote Send/Receive/Reply mechanism by acting as surrogate pairs in the remote communication. When surrogate\_R starts up it forks a surrogate\_r process. After a successful fork, surrogate\_R reports the SIMPL name of the surrogate\_r process to the protocol router and the protocol router adds this SIMPL name to an internal table.

Upon receiving a remote name locate request from a prospective sender application, protocol router checks its internal table for a surrogate\_r with the correct protocol (or in the default case the first available surrogate\_r in the table) and in the case of a match, replies the SIMPL name of the surrogate\_r process.

There exists a SIMPL utility program called *dumpProtocolTable*. When run from the command line it sends a message to the protocol router program asking for the contents of its internal table. Upon receipt of this information, dumpProtocolTable displays a list of the various surrogate\_r programs available. It is run as is and takes no command line parameters. It is a good way to check that the required surrogates are available for use.

### 7.4.1 Name Locate Operations

The *name\_locate* call performs the following actions internally:

1. Deciphers the input string. This action identifies the nature of the communication as to whether it is local or remote. If local, then the name of the receiving process is all that is important and anything else is discarded. If remote, then the name of the receiving process and the host computer are essential. If the protocol is not present, then the default protocol will be tried.
2. If the name located process is local, then a connection is made to its trigger FIFO and we are finished. If the process is remote, then the local protocol router process is name located.
3. Once the protocol router has been name located a message is sent to it requesting the name of the next surrogate\_r process with the desired protocol that is available. The protocol router provides this information in a reply message.
4. When the reply message with the necessary information, the local surrogate\_r is name located and the remote name locate mesage is sent to it. The surrogate\_r process then tries to make a connection to the remote host and finally to its surrogate\_s partner if all goes well.
5. The surrogate\_r program replies back the result of the remote name locate and the name locate function returns with the SIMPL ID of the surrogate\_r process or a failure.

## 7.5 Surrogates and Protocols

While surrogates facilitate transparent transport of SIMPL messages between SIMPL processes on different network nodes, they also introduce complications associated with failures at the network transport layer. An ideal network transport layer would be capable of detecting and reporting network failures in a timely manner to the respective surrogates at each end. Once such a network failure is detected the challenge is to report such failures back to the actual SIMPL application in a manner consistent with the *need to know* SIMPL axiom. Recall Section 3.2 in Chapter 3.

The SIMPL sender should not need to know that a network transport layer is being used to transport SIMPL messages to the intended SIMPL receiver.

The TCP/IP surrogate is the original SIMPL surrogate and hence is widely used. TCP/IP as a network transport layer is far from ideal when it comes to detecting and reporting network or socket failures.<sup>1</sup>

In fact, TCP/IP cannot easily detect the loss of connectivity with a socket partner without the use of a separate heartbeat packet and a timeout. This poses immense challenges for the design of a TCP/IP surrogate. While a heartbeat/-timeout combination works reasonably well for outright network failures, it can yield false positives on a sluggish TCP/IP network. Such a failure is communicated back to the SIMPL sender in an error message which is indistinguishable from a local receiver failure. If care isn't taken with the error recovery algorithm, runaway loops can result especially if the algorithm is a straight retry.

The bottom line is that while the SIMPL developer is largely isolated from the details of network transport, it is important that the developer be aware of the failure detection limitations associated with the particular network transport layer planned for the application.

### 7.5.1 TCP/IP Surrogate

TCP/IP surrogates were the first to be written because of the pervasiveness of these protocols. Based on a layering model that has met with success in the development of network protocols, TCP/IP allows connectivity between all sorts of computer hardware running under different operating systems and as such has made possible the Internet.

For the operational details of the TCP/IP surrogate, please refer to Appendix H.

### 7.5.2 RS-232 Surrogate

The RS-232 surrogates operate somewhat differently from the TCP/IP surrogates. From a remote application's point of view there is no difference of course. Because there are no sockets ultimately attached to the serial port there is only one conduit available. In this way, host one attaches to host two via one serial line. Because of the surrogates, any number of programs on host 1 may communicate with any number on host 2, however communications with any other host would require another serial line. An exception to this would be the case that a computer host communicated with a number of devices all connected to

---

<sup>1</sup>TCP/IP Illustrated, Volume 1 by W. Richard Stevens, Addison-Wesley Pub.,1994, ISBN 0-201-63346-9.

the same serial line. All of the devices would be privy to all communications but would be programmed to only respond to some unique identification. This would be a good example of where tokenized message passing would be valuable. Refer to Chapter 5 for more on tokenized messages.

For the operational details of the RS-232 surrogate, please refer to Appendix I.

### 7.5.3 tclSurrogate

The SIMPL toolkit has always been useful for constructing applications on Linux (or other UNIX like) systems. Very early on it became apparent that there was a need to construct hybrid applications where SIMPL message exchange would occur between a SIMPL process and code running on a non-UNIX like OS.

The first such hybrid application for SIMPL involved message exchanges between a Linux server running SIMPL and a Windows client running a Tcl/Tk applet. The design objective was to preserve the Send/Receive/Reply aspect of SIMPL messaging in this hybrid environment. The solution was to divide the Windows client process into two parts:

- the SIMPL messaging part
- the business logic part

These two parts were then connected using a TCP/IP socket. The tclSurrogate protocol (see Appendix J) was then used to facilitate the communication between the parts. Once this separation was envisioned it became possible to separate the SIMPL part and the business logic part on separate network nodes.

As the design progressed it became apparent that the SIMPL portion could be made entirely generic. It was designed as a forking Linux daemon which was given the name tclSurrogate. This tclSurrogate was designed to listen to port 8000 and fork a child process upon connection. This child process then became the SIMPL part of the remote application. Since the early remote applications were written in Tcl/Tk the name tclSurrogate was chosen and has stuck ever since.

To facilitate the building of Tcl/Tk applications using this tclSurrogate gateway, a Tcl/Tk package (library) was constructed. The early version of this library (fcsocket.tcl) simply exposed the raw TCP/IP tclSurrogate protocol as a series of functions. This worked well, but the resulting Tcl code did not look like the SIMPL API.

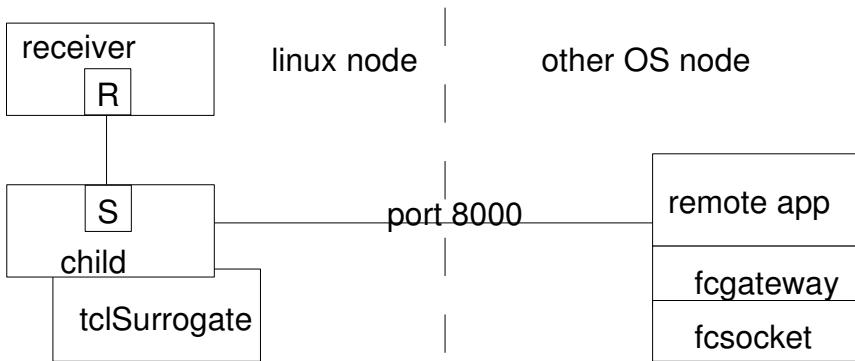


Figure 7.2: tclSurrogate

Meanwhile, work started on a Tcl/Tk shared library for SIMPL (fctclx). Here the Tcl/Tk language was extended to include the SIMPL API. The shared library was written in ‘C’ and hence was able to call the SIMPL core library directly. At this point Tcl/Tk standalone SIMPL applications could be written for Linux systems, ie. systems which could support the Tcl/Tk shared library. This Tcl/Tk API is described in Appendix G. At this point it was possible to write SIMPL applications in Tcl/Tk using two different approaches:

- the tclSurrogate gateway approach
- the Tcl/Tk shared library approach

However, the code did not look the same. An abstraction layer library (fc-gateway) was created in Tcl/Tk to remap the Tcl/Tk shared library API calls onto the raw tclSurrogate calls. Today Tcl/Tk applications can be written to a single unified API and deployed either as a stand alone SIMPL application (Linux) or as a tclSurrogate gateway connected app (Windows).

Meanwhile the tclSurrogate protocol proved to be quite flexible. SIMPL now supports tclSurrogate protocol libraries for Python and Java. The tclSurrogate protocol was adapted to other hybrid SIMPL systems. One of these involved adapting this protocol to a deeply embedded network appliance.<sup>2</sup> This allowed for totally transparent SIMPL messaging to occur between a Linux server and this network appliance or between a standalone Tcl, Python, or Java application and this network appliance.

---

<sup>2</sup>A paper describing the use of the tclSurrogate protocol for a deeply embedded network appliance is available at: <http://www.hometoys.com/htinews/apr06/articles/appliance/part2.htm>.

### 7.5.4 QNX Surrogates

A number of years ago it became necessary for SIMPL to communicate with QNX. QNX performs its network Send/Receive/Reply on the network level. At the time, only the TCP/IP surrogate existed for SIMPL. So how does one get a Linux host to talk to a QNX host at the application level? We decided to add TCP/IP surrogates to QNX. They run almost exactly the same as those on Linux but the local core Send/Receive/Reply calls used are strictly QNX's own.

The QNX version supported is 4.25 which is rather dated now so we only mention this in passing. Current versions of SIMPL still communicate with those surrogates written for QNX4.25. The QNX surrogate software is available on the SIMPL website.

### 7.5.5 Internal Parameters

There are a couple of internal parameters for configuring SIMPL surrogates. They are as follows:

1. Message Type. This is discussed in detail in Section 7.5.6. This has to do with the byte ordering of integers and how different computer architectures store integers in memory. If all of the networked hosts follow the same scheme then we may compile the surrogates as follows:

**make install MSG\_TYPE=SUR\_BIN**

This is currently the default. If on the other hand the various hosts store integers differently, the the surrogates should be compiled as follows:

**make install MSG\_TYPE=SUR\_CHR**

2. Message Buffering. This only applies to TCP/IP surrogates and not to RS-232 surrogates. If message buffering is desired for reading and writing, it can be enforced. For message buffering the TCP/IP surrogates must be compiled with the following directive:

**make install MSG\_BUFFERING=BUFFERED**

Messages are not buffered by default. The default directive for no buffering is as follows:

**make install MSG\_BUFFERING=NOT\_BUFFERED**

### 7.5.6 Endian Issues

Different computer architectures may store information differently. One of these differences may be the order of storage in terms of memory addressing otherwise

AB	CD	Big Endian
CD	AB	Little Endian

Table 7.1: Endian Memory Storage of 0xABCD

known as byte ordering. Suppose by way of an example we have a short integer. This consists of two bytes. Suppose further that we have the number 43981 which is ABCD in hexadecimal. The following table shows how this number could be stored.

If a big endian host sent this integer to a little endian host, the number would appear to be CDAB or 52651 in decimal. If a little endian host sent this integer to a big endian host the number would appear also to be CDAB.

If the various hosts that are using SIMPL on a network are homogeneous, ie. they are all big endian or little endian then there is no problem. The difficulties arise when there is a mix of computer types. This was a problem within socket programming due to the heterogeneous nature of networks like the Internet. A discussion of network byte ordering is beyond the scope of this book but how it affects SIMPL is not. In terms of the Internet, big endian was chosen as the de facto standard for integer exchanges. The issue is overcome in that all integers are converted from little endian to big endian if necessary prior to communication and then turned back into little endian as the case may be. This is usually known as *host ordering to network ordering* and *network ordering to host ordering* respectively.

Networked SIMPL uses tokenized message passing between surrogates in order to identify the sorts of messages being passed. See Appendix K. These tokens are integers and so could also fall into the same difficulty between heterogeneous hosts. Rather than follow the method of converting integers from little endian to big endian and vice versa, the SIMPL internal tokens are converted to character strings prior to sending and converted back upon reception.

SIMPL currently uses the binary approach by default. If you are using SIMPL on a heterogeneous network then the surrogates must be forced to use the character string approach. For SIMPL to use the character approach the surrogate software must be compiled with the following directive:

**make install MSG\_TYPE=SUR\_CHR**

**IMPORTANT:** As far as SIMPL is concerned, the message content is of no interest. The nature of the content is beyond the scope of SIMPL since a message can be anything at all and its true nature is only understood by the sender and the receiver. Consequently, what happens to the message is up to the sender

Binary to Character	Character to Binary
<code>btosUI(unsigned int n, char *s, int c)</code>	<code>stobUI(char *, int c)</code>
<code>btosSI(signed int n, char *s, int c)</code>	<code>stobSI(char *, int c)</code>
<code>btosUSI(unsigned short int n, char *s, int c)</code>	<code>stobUSI(char *, intc)</code>
<code>btosSSI(signed short int n, char *s, int c)</code>	<code>stobSSI(char *, int c)</code>
<code>btosF(float n, char *s, int c)</code>	<code>stobF(char *, int c)</code>

Table 7.2: Binary  $\iff$  Character Conversion Functions

and the receiver to sort out. If you are sending integer or floating point data etc. it will be necessary to re-interpret the data correctly.

There are some functions available for converting various ‘C’ variable types to a character format and vice versa within the *simplmisc* library. These functions are shown in Table 7.2. The *btos* functions are of type void and the *stob* functions return the number type desired. The *int c* appearing in each function is the width of type *int* on the host in question. For example, on a 32-bit system the int width is 4.

## 7.6 Summary

In this chapter we covered a great deal of material. The most important points to remember are:

- Surrogates are programs that occur in sender/receiver pairs and communicate with each other remotely via some communication protocol.
- Networked SIMPL is accomplished by the use of surrogate pairs.
- Surrogate pairs currently utilize either TCP/IP or RS-232 communication protocols.
- Required programs to enable the use of surrogates are the `protocolRouter`, `surrogateTcp` for TCP/IP, and `surrogateRS232` as well as `rs232_rw` for RS-232.
- There are some specialized surrogates also available for Tcl and for QNX.

## 7.7 Next Chapter

The next chapter is about a recent and ongoing project to set some of SIMPL's core elements within the Linux kernel. It is a self-contained chapter that can be skipped entirely without losing any continuity. That would be shame however, since it is an interesting subject.

# Chapter 8

## Raspberry Pi and Arduino

In this chapter we are going to make use of some of what we've learned so far. We are going to make a program that runs on an *Arduino* microcontroller and SIMPL communicates with another program running on a *Raspberry Pi* computer.

### 8.1 Raspberry Pi

The Raspberry PI is an inexpensive credit card sized embedded Linux computer developed in the UK by the Raspberry Pi Foundation.<sup>1</sup> While the Raspberry Pi was conceived and developed without SIMPL in mind (at least to our knowledge), it is an ideal platform to host all or a portion of a SIMPL app. Not only can a Raspberry Pi node host both flavors of SIMPL surrogate daemons it can run native SIMPL modules just like any Linux node. Through the use of the tclSurrogate daemon the Raspberry Pi can be an ideal interface to the I/O on a deeply embedded Arduino.<sup>2</sup> SIMPL supports all the Raspberry Pi programming languages including the most popular Python language. SIMPL-Python is a full feature implementation of SIMPL. However, that said, SIMPL applications can be built up of modules written in mixed languages, so C could be used for modules where performance is an issue.

Since the Linux process scheduler is not considered real time, so is SIMPL not real time. However through careful allocation of critical modules to a dedicated processor such as the Raspberry Pi it is often possible to improve response times

---

<sup>1</sup><http://www.raspberrypi.org>

<sup>2</sup><http://www.raspberry-pi-geek.com/Archive/2014/06/Joining-Raspberry-Pi-and-Arduino-applications-with-SIMPL>

compared to those on a shared multipurpose Linux server. With the Raspberry Pi cost point this is even more feasible on many projects. Through the use of SIMPL surrogates it is very easy to distribute the SIMPL modules of a given SIMPL app onto one or more Raspberry Pi nodes.

Given Raspberry Pi's small size and low cost, it is ideally suited for use in embedded projects, educational purposes and hobbyist projects. They are based on a SoC (System on a Chip) design which is essentially an IC (Integrated Circuit) which combines many or all of a computer's components onto a single chip.

## 8.2 Arduino

Arduino<sup>3</sup> is a computer hardware and software company that provides various microcontroller boards that can be programmed via an IDE (Integrated Development Environment). This IDE can be run from various different host computers for program development and loading executable programs into the Arduino's flash memory. In our case, we are using a Linux host.<sup>4</sup> The Arduino boards themselves have various names along with a varied functionality and are often referred to simply as *Arduinos*. Like the Raspberry Pi, they are credit card sized and low in cost and are ideally suited for embedded projects. Unlike the Raspberry Pi however, they have no operating system and so they usually run one single dedicated program. They come with a number of programmable I/O ports and a set of hardware modules called *shields* which are PCBs (Printed Circuit Boards) which plug into the Arduino microcontroller board and extend its function. The SIMPL example we use later in this chapter uses an Arduino *Uno* with an *Ethernet Shield* so that the Arduino can communicate over an Ethernet network. Other available shields provide motor control, LCD screens, and so on. Arduinos are a wonderful learning tool and great fun to experiment with.

In Chapter 7, we discussed networking with SIMPL and the concept of surrogates. Among the surrogates, we discussed the *tclSurrogate* in section 7.5.3. The *tclSurrogate* was originally contrived to allow SIMPL messages to be routed between Unix-like hosts and non-Unix-like hosts, say Linux and Windows. When the opportunity arose to warrant the use of an Arduino on a real life project, SIMPL messaging was very much desired. The question was however: How do we run surrogates along side of an application program on an Arduino, given that there is no OS? The answer is: Use a *tclSurrogate*. Why? Because the *tclSurro-*

---

<sup>3</sup><http://www.arduino.cc/>

<sup>4</sup>The Arduino IDE also runs on Windows, MAC OS X and others.

gate can reside on a different platform than the SIMPL sender or receiver running on the Arduino. Since the Arduino can only run a single dedicated program at a time, the tclSurrogate protocol is a good fit. The actual sender/receiver program can run on the Arduino and the tclSurrogate can run on a computer host elsewhere in a connected network.

There is a ‘C’ library which allows SIMPL messaging between Arduinos and computer hosts connected on the same network. See Appendix D for details on this library.

## 8.3 Hangman Game

The *Hangman* game is thought by some to go back to the 1800’s. It is a game where one person thinks up a word and another person or persons try to guess the word by guessing the word’s characters one at a time. Each correct guess puts a character into the word. Each incorrect guess adds another piece to the hangman picture. The mystery word must be guessed before the picture is complete or you lose.

Figure 8.1 is an example of a lost game. Can you guess the word?

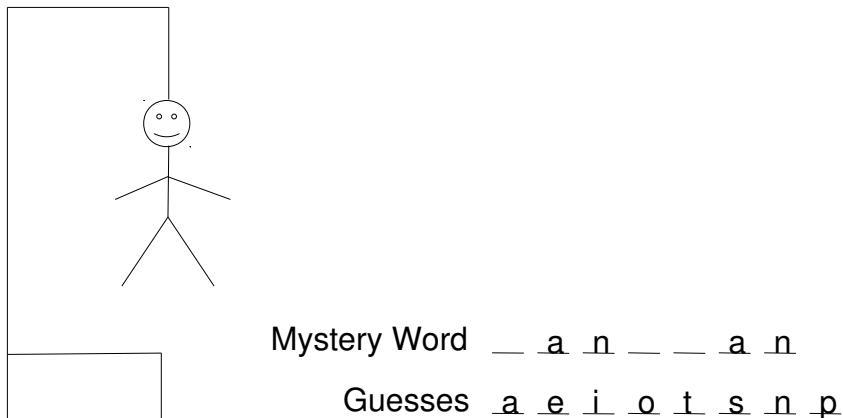


Figure 8.1: Hangman Game

### 8.3.1 Hangman Design

We have elected to break up the hangman game in the following manner, see Figure 8.2.

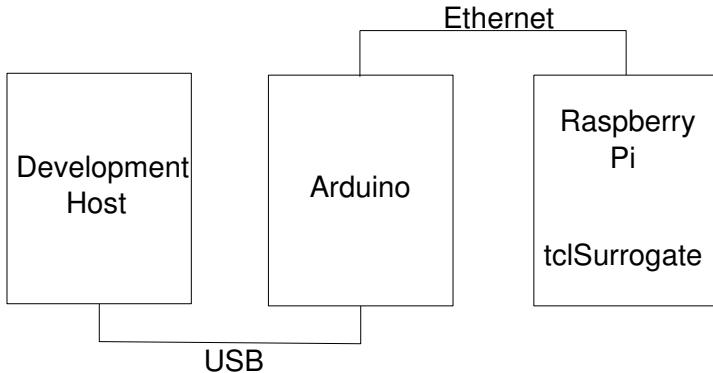


Figure 8.2: Game Network

Firstly, we have a development computer on the far left of Figure 8.2. This computer is running Linux and provides the IDE to the Arduino and is connected to it via a USB cable. The Arduino IDE, among other things, provides a way to edit program files, compile and link them, and then load the resulting executable into the Arduino's memory for execution. For our hangman game we use this computer to write and load the hangman program for the Arduino and via the *serial monitor* within the IDE, receive visual feedback from the Arduino in the form of the game interface. As well, guesses for the word characters will come via the development computer's keyboard.

Secondly, we have the Arduino itself. It will run a hangman program which communicates via the *serial monitor* interface with the development computer. The hangman program will render the game interface on the development computer's monitor and accept *guesses* (alphabetic characters) from the development computer's keyboard. When it needs a mystery word, it will send a SIMPL message to a program running on the Raspberry Pi requesting such a word. Given that the Arduino uses the tclSurrogate protocol for SIMPL communications, it will SIMPL *name attach* on the Raspberry Pi which is running the tclSurrogate program.

Thirdly, on the far right of Figure 8.2 we have the Raspberry Pi. It is running a SIMPL receiver program that receives requests for a mystery word. It looks up a word at random in an English dictionary file and replies the word back to the sender. This program has been written in Python and is compatible with Python 2.\* and 3.\*. To enable SIMPL networking between the Raspberry Pi and the Arduino, the tclSurrogate program is running on this host. It is enabled

by entering the following command: `tclSurrogate -a 50005` & on the command line, where 50005 represents the port number that the tclSurrogate is listening on.

### 8.3.2 SIMPL Receiver

The receiver for our purposes is a Python script and is running on the Raspberry Pi.

*hangman.py*

```
1 #import required modules
2 import csimpl
3 import random
4
5 WORD_TOKEN = 42
6 MAX_LEN = 10
7 MIN_LEN = 4
8
9 def deconstructReceiveMsg(nee):
10     # extract the incoming message
11     token = nee.unpackMsg(nee.SIMPL_BIN, "i")
12     if token == WORD_TOKEN:
13         return True
14     else:
15         return False
16
17 # does the randomly chosen word fit the desired criteria?
18 def checkWord(w):
19     length = len(w)
20     if length > MAX_LEN or length < MIN_LEN:
21         return False
22     # length - 1 because the final character is '\n'
23     for i in range(0, length-1):
24         if w[i].isupper() or not w[i].isalpha():
25             return False
26     return True
27
28 def constructReplyMsg(nee):
29     # open the file for read
30     f = open("/usr/share/dict/linux.words", "r")
31
32     while (True):
33         # get a random number for offset to dictionary file
34         offset = random.randrange(100, 478932)
35
```

```

36     # finally read that particular line/word
37     for line in range(0, offset):
38         word = f.readline()
39
40     # word is OK?
41     if checkWord(word):
42         break
43
44     # start at the beginning again
45     f.seek(0)
46
47     # close file
48     f.close()
49
50     # load reply message into the outgoing message buffer
51     nee.packMsg(nee.SIMPL_BIN, "iis", WORDTOKEN, len(word), word)
52
53 ##### main part of program ****
54
55 # constructor for simpl class object
56 nee = csimpl.Simpl("HANGMANPY", 1024)
57 while 1:
58     # receive a message
59     messageSize, senderId = nee.receive()
60     if messageSize == -1:
61         print("receive error")
62         continue
63
64     # examine the message
65     if deconstructReceiveMsg(nee):
66         # build reply message
67         constructReplyMsg(nee)
68         # reply to sending program
69         if nee.reply(senderId) == -1:
70             print("reply error")

```

lines 2-3 Import required modules.

lines 5-7 Set some definitions.

lines 9-15 A function for deconstructing a received message which pays particular attention to the message token.

lines 18-26 A function used for checking that the word that was randomly selected from the file of English words is suitable. (That said, we are not

responsible for the contents of the file. It was chosen because it is *open source* in origin; some of the words are quite interesting in that we have never heard of them or won't admit to it!)

lines 28-51 A function for constructing a reply to a sent message. After a valid word is chosen from the file of English words, it is then loaded into the reply message. The reply message to be sent back to the sender is made in line 51.

line 56 An instance of the csimpl Simpl object is created. The required SIMPL name attach is made within the constructor. *HANGMAN\_PY* is the SIMPL name of this SIMPL receiver Python program and *1024* is the maximum size an incoming or outgoing message can be. Note that this value far exceeds

lines 58-70 SIMPL messages are received by this script, examined, and replied to. Note that the desired *nameDetach()* call is made by the SIMPL object destructor when the program is no longer running.

### 8.3.3 SIMPL Sender

The sender for our purposes is a 'C' program running on the Arduino. Note that this program has been run successfully on the Arduino Uno and Mega.

#### hangman.c

```
1 #include <SPI.h>
2 #include <Ethernet.h>
3 #include <Simpl.h>
4
5 // definitions
6 #define MAXLEN    10
7 #define ASTERISK   '*'
8 #define SPACE      ' '
9 #define WORD_TOKEN 42
10 #define YES        'y'
11 #define NO         'n'
12
13 // various stages of the puzzle
14 typedef enum
15 {
16     START,
17     HEAD,
```

```
18     BODY,
19     LEFT_ARM,
20     RIGHT_ARM,
21     LEFT_LEG,
22     DEAD
23 } STAGE;
24
25 typedef enum
26 {
27     UNKNOWN,
28     LOSE,
29     WIN
30 } OUTCOME;
31
32 typedef struct
33 {
34     long token;
35     long wordLength;
36     char word[MAX_LEN];
37 } MSG;
38
39 // function prototypes
40 void hangman(void);
41 void draw(int);
42 void verbage(int);
43 char getEntry(void);
44 boolean doHangStr(char);
45 boolean doGuessStr(char);
46 void checkStage(void);
47 char getRepeat(void);
48
49 // global variables
50 EthernetClient client;
51 long receiverId;
52 char wordStr[MAX_LEN];
53 char hangStr[MAX_LEN];
54 char guessStr[2 * MAX_LEN];
55 int stage;
56 int outcome;
57
58 void setup()
59 {
60     const unsigned port = 50005;
61     char *server = "192.168.1.10";
62     byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77};
63     unsigned char ip[] = {192, 168, 1, 11};
```

```
64 // start serial messaging to a local monitor
65 Serial.begin(9600);
66 Serial.println();
67 Serial.println("Starting up.");
68
69 // start ethernet
70 Ethernet.begin(mac, ip);
71 // give the ethernet shield hardware an opportunity to initialize
72 delay(1000);
73
74 // remotely SIMPL name attach to the designated server and port
75 if (nameAttach(server, port, &client, "HANGMAN") == -1)
76 {
77     Serial.print("name attach error: ");
78     Serial.println(whatsMyError());
79     client.stop();
80     while(1);
81 }
82
83 // SIMPL name locate the receiver process desired for messaging
84 receiverId = nameLocate(&client, "HANGMANPY");
85 if (receiverId == -1)
86 {
87     Serial.print("name locate error: ");
88     Serial.println(whatsMyError());
89     nameDetach(&client);
90     client.stop();
91     while(1);
92 }
93
94 Serial.println("Initialization is complete.");
95 }
96
97 void loop()
98 {
99 // play hangman
100 hangman();
101
102 // game's over
103 Serial.println();
104 Serial.println("Game Over ... going to spin now");
105 Serial.println();
106
107 // spin
108 while (true);
```

```
110  }
111
112 /*——————*/
113
114 void hangman()
115 {
116 // client is global
117 // 'stage' is global
118 boolean match1 = true;
119 boolean match2 = true;
120 MSG msg;
121 int ret;
122 int mSize = sizeof(MSG);
123
124 msg.token = WORD_TOKEN;
125
126 do
127 {
128 // initialize some variables
129 memset(wordStr, NULL, MAXLEN);
130 memset(guessStr, NULL, 2 * MAXLEN);
131 memset(hangStr, NULL, MAXLEN);
132 stage = START;
133 outcome = UNKNOWN;
134
135 // get a hangman word via SIMPL
136 ret = Send(&client, &receiverId, &msg, &mmsg, mSize, mSize);
137 if (ret == -1)
138 {
139     Serial.print("Send error ");
140     Serial.println(whatsMyError());
141     break;
142 }
143
144 // set the word, don't want the last character
145 memcpy(wordStr, msg.word, msg.wordLength - 1);
146 memset(hangStr, ASTERISK, strlen(wordStr));
147 memset(guessStr, SPACE, 2 * MAXLEN - 1);
148
149 // draw the starting hangman picture
150 draw();
151
152 // print up the start wording
153 verbage();
154
155 do
```

```
156     {
157     // obtain guessed character from player , check validity
158     char guess = getEntry();
159
160     // does the hangStr have this character?
161     match1 = doHangStr(guess);
162
163     // has this character been guessed before?
164     match2 = doGuessStr(guess);
165
166     // adjust the stage?
167     if ( !match1 && !match2 )
168         stage += 1;
169
170     // draw the hangman picture
171     draw();
172
173     // print up the wording
174     verbage();
175
176     // check the stage we are at
177     checkStage();
178     }
179     while (outcome == UNKNOWN);
180   }
181 while (getRepeat() == YES);
182
183 nameDetach(&client);
184 }
185
186 /*——————*/  

187
188 char getEntry()
189 {
190 // retrieve a typed in guess character and check its validity
191 char guess;
192
193 while (true)
194   if (Serial.available())
195   {
196     guess = Serial.read();
197     // a valid entry?
198     if (isalpha(guess))
199       break;
200   }
```

```
202 return guess;  
203 }  
204  
205 /*—————*/  
206  
207 boolean doHangStr(char ch)  
208 {  
209     boolean match = false;  
210  
211     // search wordStr replacing matches in hangStr with ch  
212     for (int i = 0; i < strlen(wordStr); i++)  
213         if (wordStr[i] == ch)  
214             {  
215                 hangStr[i] = ch;  
216                 match = true;  
217             }  
218  
219     return match;  
220 }  
221  
222 /*—————*/  
223  
224 boolean doGuessStr(char ch)  
225 {  
226     // character already been guessed?  
227     for (int i = 0; i < strlen(guessStr); i++)  
228         if (guessStr[i] == ch)  
229             return true;  
230  
231     for (int i = 0; i < strlen(guessStr); i++)  
232         if (guessStr[i] == SPACE)  
233             {  
234                 guessStr[i] = ch;  
235                 break;  
236             }  
237  
238     return false;  
239 }  
240  
241 /*—————*/  
242  
243 void checkStage()  
244 {  
245     // 'stage' is global  
246     // have you come to the end of the road?  
247     if (stage == DEAD)
```

```
248     {
249         Serial.println();
250         Serial.print(" You Lose!    Word was: " );
251         Serial.println(wordStr);
252         outcome = LOSE;
253     }
254
255 // are there any asterices left in the hangStr?
256 if (strchr(hangStr, ASTERISK) == NULL)
257 {
258     Serial.println();
259     Serial.println(" You Win!");
260     outcome = WIN;
261 }
262
263 /*
264 */
265
266 char getRepeat()
267 {
268     Serial.println();
269     Serial.println(" Do you want to try another? y/n" );
270     return getEntry();
271 }
272 /*
273 */
274
275 void verbage()
276 {
277 // print the word characters
278     Serial.print(" Word: " );
279     Serial.println(hangStr);
280     Serial.println();
281     Serial.print(" Guesses: " );
282     Serial.println(guessStr);
283 }
284 /*
285 */
286
287 void draw()
288 {
289 // 'stage' is global
290 // draw the hangman picture w.r.t. the various stages
291     Serial.println("      ----" );
292     Serial.println("      |      |" );
293     switch (stage)
```

```

294 {
295     case START: // start
296         Serial.println(" | ");
297         Serial.println(" | ");
298         Serial.println(" | ");
299         break;
300     case HEAD:
301         Serial.println(" |     O");
302         Serial.println(" |   | ");
303         Serial.println(" |   | ");
304         break;
305     case BODY:
306         Serial.println(" |     O");
307         Serial.println(" |   | ");
308         Serial.println(" |   | ");
309         break;
310     case LEFT_ARM:
311         Serial.println(" |     O");
312         Serial.println(" |   /|");
313         Serial.println(" |   | ");
314         break;
315     case RIGHT_ARM:
316         Serial.println(" |     O");
317         Serial.println(" |   /|\\" );
318         Serial.println(" |   | ");
319         break;
320     case LEFT_LEG:
321         Serial.println(" |     O");
322         Serial.println(" |   /|\\" );
323         Serial.println(" |   / ");
324         break;
325     case DEAD:
326         Serial.println(" |     O");
327         Serial.println(" |   /|\\" );
328         Serial.println(" |   / \\\" );
329     }
330     Serial.println(" | -----");
331     Serial.println(" |       | ");
332     Serial.println();
333     Serial.println();
334 }
```

lines 1-3 Include required header files.

lines 6-30 Define some useful constants.

lines 32-37 Define the message that will be sent to the hangman receiver program and replied back. Note, often the sent and received message is different in content from the message that is replied back to the sender. The same message *template* has been used for both for the sake of simplicity.

lines 40-47 Function prototypes are defined.

lines 50-56 Global variables are defined.

lines 58-96 The necessary `setup()` function is defined.

lines 60 This is the port that the tclSurrogate program is babysitting.

line 61 This is the IP address of the Raspberry Pi on this network.

line 62 This is the MAC address of the Ethernet Shield attached to the Arduino.

line 63 This is the IP address we want to assign to the Arduino on this network.

line 66 The serial monitor is initialized. This is the way that the Arduino displays images on the development host's monitor and gathers input in the form of characters from the development host's keyboard.

line 71 The Ethernet shield is initialized thus allowing a networking between other computer hosts and the Arduino. In this case, an IP has been chosen as *192.168.1.11* rather than using DHCP.

lines 76-82 The hangman sender program SIMPL *name attaches* via the tclSurrogate running on the Raspberry Pi with IP *192.168.1.10*.

lines 85-93 The hangman sender program SIMPL *name locates* the hangman receiver program which is SIMPL name attached as *HANGMAN\_PY*.

lines 98-110 The necessary `loop()` function is defined.

lines 114-184 The *working* function of the program, not surprisingly called *hangman()*.

lines 136-142 Send a message to *HANGMAN\_PY* via the tclSurrogate asking for a mystery word. Extract the mystery word from the reply message and start the game.

lines 188-334 The auxiliary functionality responsible for drawing the hangman picture at its various stages and getting input from the player(s) via the keyboard.

### 8.3.4 What you should see on the Development Host

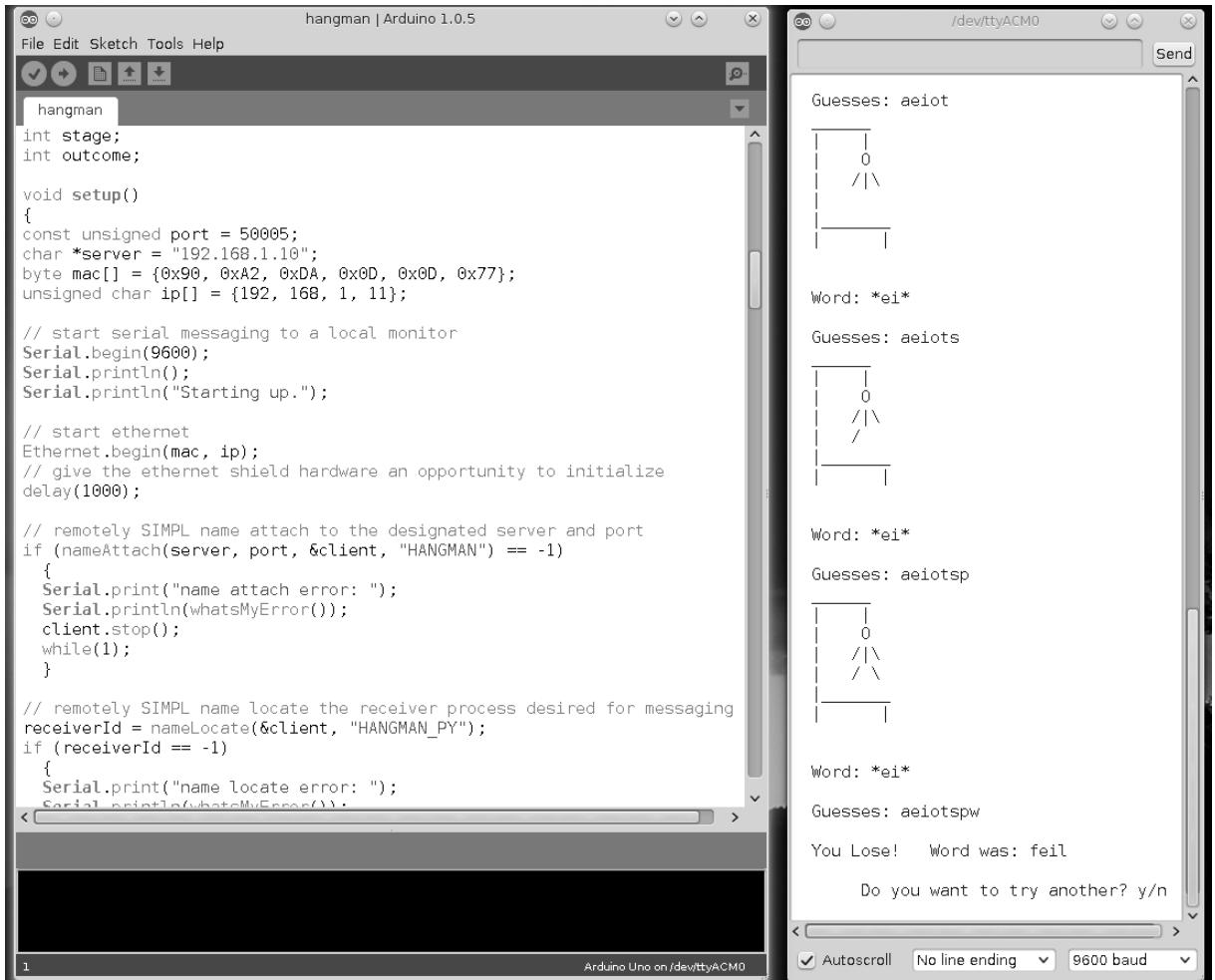


Figure 8.3: Hangman Screen

Figure 8.3 shows what should be seen on the development host's screen. The window on the left is the Arduino development IDE where the C program called `hangman.ino` is being edited, compiled and uploaded to the Arduino for execution. The window on the right provides the interface to the game via the serial monitor, also within the IDE. This is where your guesses (characters) are uploaded to the Arduino and then your relative success or failure is made apparent.

## 8.4 Summary

In this chapter we have discussed using SIMPL messaging between a Raspberry Pi computer running Linux and an Arduino microcontroller board. The main reason for this is to demonstrate how SIMPL messaging can be used on an embedded scale. The following are the main points of this chapter:

- SIMPL can run on a small scale host like the Raspberry Pi.
- SIMPL can run on an Arduino via the tclSurrogate protocol.
- SIMPL can *glue* embedded devices to a much larger world.

## 8.5 Next Chapter

In the next chapter we are going to begin discussing the importance of testable code as good programming practice and how this concept is furthered and made easier by the use of SIMPL.



# **Part III**

# **SIMPL Toolkit**



# Chapter 9

## Testing Framework

### 9.1 What is a Testing Framework?

Almost all real world problems that the SIMPL toolkit would be used to solve are complex. We saw in Chapter 3 on softwareICs that SIMPL promotes the decomposition of complex problems into islands of manageable complexity. We also saw in Chapter 5 that those SIMPL modules are readily extendable. None of this would matter much if the SIMPL modules and the SIMPL application were not readily testable.

Author Kent Beck's book<sup>1</sup> on the Extreme Programming development approach expounds at length on the importance of testable code in improving project success metrics. In the ideal Extreme Programming project environment the culture is such that tests are created before the code is written. While this ideal is difficult to maintain in practice, we are going to show that the SIMPL testing framework (STF) promotes good coding practices.

- Testable code is an integral part of any successful software development methodology.
- Testable code requires the ability to stimulate all the execution paths without disturbing the executables to do the stimulation.
- Testable code is greatly facilitated if the toolkit under which the application was developed allows for complexity encapsulation, preferably as independent modules.

---

<sup>1</sup>Extreme Programming Explained by Kent Beck, Addison-Wesley Pub., 2000, ISBN 0-201-61641-6

- Testable code is easier if the test tools can be written in the best language for the task rather than just the language of the application.
- If the application being tested is going to be deployed across a network, testable code dictates that unit tests can be run without the network.
- Testable code is enhanced if the execution sequences are always deterministic and predictable. More importantly, this determinism persists even if any particular section of the application comes under load.
- If the developer can extend functionality in the application with a low probability of affecting existing functionality, testability increases.

Before moving on allow us to digress. Much is written and said about the need to document code. We encounter this often in our consulting practice. One of the stated purposes of good documentation is to reduce the learning curve for a new developer charged with the repair or extension of a software application once the original development team has been disbanded. While documentation is important, we believe that it is overrated as a cost effective means to reduce such learning curves. The production and maintenance of a good testing framework is a much more cost effective way to jump start the second team developer. Furthermore, the deployment of an effective testing framework will also lower the costs associated with feature delivery in the original developer team. Like documentation however, unless the original team embraces the concept of a testing framework from the outset, it will be diminished in effectiveness. Alas, retrofitting a good testing framework after the fact is as difficult to achieve as retrofitted documentation.

In this chapter we are going to show how the SIMPL toolkit makes testable code easier. While we are at it, we are going to illustrate the SIMPL design methodology.

In many programming shops, testing (or QA) is viewed as a separate activity from code design and programming. To make matters worse, this separation of responsibility often leads to separate toolsets for programmers and for testers. This separation is unfortunate because testable code needs to be designed into the project from the outset. By definition the software developer needs to be intimately involved.

*Software developers hate testing.*

This presents a challenge; how do you get developers to adopt a testable code approach?

You have to keep things simple and you have to:

1. Make the testing framework out of the same toolkit primitives used by the developer (eg. text files, scripts, symbolic links etc.).
2. Demonstrate convincingly that the added effort required to build and maintain this testing framework is more than offset by the reduction in time to market for mainline code.

The SIMPL project knows this all too well. Many of the SIMPL testing framework (STF) ideas co-evolved with the SIMPL codebase itself. As a result we've had to migrate developer test scripts over to the new framework after the fact and this is far from complete. Having said this, the STF has been battle tested on a number of SIMPL projects that we have been involved with. When properly deployed it works.

Tests are usually a painful exercise for developers. SIMPL developers are no exception. Unless you make the toolset quite simple and yet powerful they will tend to avoid this exercise. To promote its use, any testing framework must be accessible from the same environment that the developer is using to create code. A successful testing framework must also be written in a language that the developer is already using or can readily master. Despite the best of intentions, all testing frameworks by definition will need to be customized and extended to meet specific needs. Finally, the project culture needs to promote the creation of the *test tools first* before a line of code is created. SIMPL can't address the project culture but the STF can facilitate such a culture when it does exist.

The STF thinks of all software tests as being decomposable into three distinct stages:

1. The setup of environment, database tables, configuration files and so on.
2. The actual execution of the test.
3. The examination of the results.

The STF maps these three steps onto three scripts organized in a simple file tree under the environment variable TEST\_HOME (see Figure 9.1).

While in simpler projects one could reasonably manage such a file tree by hand, it rapidly becomes tedious on large real world projects. Tedium is a disincentive for promoting the testing framework's uptake into the project culture.

As such the STF has evolved a series of PATH located wrapper scripts which can be executed from any location in the developer tree. In Figure 9.1 we see three of these wrapper scripts illustrated for a fictional test 1234.

`$TEST_HOME/testing/test1234/`

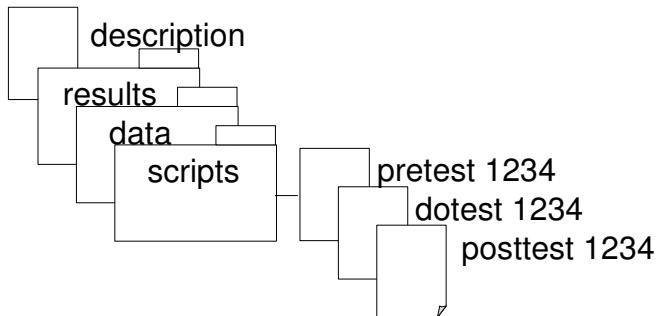


Figure 9.1: SIMPL Test Framework

Let us illustrate the STF by way of a progressively more complex example. The SIMPL project maintains the elements of the SIMPL testing framework as a separate tarball. The installation is described on the project website and also in Appendix A. Once installed the STF should have populated the `SIMPL_HOME/scripts` area with a series of wrapper scripts with names like `seetest`, `pretest`, `posttest`, `copytest` etc. If you take a look at those wrapper scripts you will find three scripts which align themselves very well to these three stages described previously:

1. `pretest` - The setup of the testing environment.
2. `dotest` - The actual execution of the test.
3. `posttest` - The examination of the results.

The degree to which each of these scripts is used depends on the project itself. In simpler projects the configuration is all self contained within the `dotest` script. Similarly, the examination (`posttest`) stage often consists of a simple observation of behaviour. Nonetheless, it is good practice to maintain and run all three scripts for each test. Experienced STF testers take advantage to make each of these scripts quite verbose in order to document the test in question more fully.

Testing, like software development, is one part creativity and five parts organization. To help with the organization part the STF is going to allow our tests to be identified by a freeform string. Most users will use a numbering scheme to classify their tests but this is not a necessary requirement for these scripts.

Another useful thing to know about the STF wrapper scripts are that they

have built in *usage* which can be accessed by simply typing the STF wrapper script name without any arguments; eg. **seetest**.

There are two ways to create a new STF test. You can cause the STF to dip into its templates to create the new test files or you can copy an existing test case. Since we are starting afresh, we haven't got any tests yet to copy so we will start by invoking the templates. Type:

```
seetest 0001
```

What should have happened is that a new subdirectory called **test0001** should have been created at **\$TEST\_HOME/testing**.

Inside that subdirectory a file called **description** should appear. The contents of that are straight text and will reflect the template at this stage. Inside a further subdirectory called **scripts** a set of templated scripts called **setup**, **runttest**, and **whathappened** should appear.

**NOTE:** In what follows we use the scripts *seetest*, *pretest*, *dotest*, and *posttest* to edit the the files *description*, *setup*, *runttest*, and *whathappened* respectively. The scripts seetest, pretest, etc. invoke the **vi** editor in order to modify the description, setup, etc. files. If you are not comfortable with *vi* or it is not available on your system, you may edit these files directly with the text editor of your choice.

The first task with a new test is usually to edit the test description file to describe the test case. Type:

```
seetest 0001 e
```

This should bring the description file up in the **vi** editor. The STF index (**seetest i**) uses the first two lines in this description file. The second line of the description file is displayed in the index so it should contain a useful summary line. The rest of the description file is free form text which can be used to document the test case covered by this test.

Not all tests will require all three stages that the STF uses. It is good practice however, to maintain each of these three scripts in any case. At a minimum the TEST\_NO and TEST\_DESC should be edited in each of these three scripts by typing:

```
pretest 0001 e  
dotest 0001 e  
posttest 0001 e
```

and making those changes to reflect the new test. You can verify that each stage can execute by typing the above commands without the "e" for edit command line argument.

We now need something concrete to add to our test. In Chapter 6 we discussed the stimulator softwareIC. This is probably one of the most frequently used (and cloned) SIMPL softwareICs because it is so useful as a sender test stub. Initially we'll focus our attention on the stage two STF script. Using the following,

**dotest 0001 e**

to edit something like:

```
#!/bin/bash

TEST_NO=0001
TEST_DESC="SIMPL book test 1"

TEST_DIR=$TEST_HOME/testing/test$TEST_NO
OUTFILE=$TEST_DIR/results/test.out

echo "Starting up test #$TEST_NO" | tee $OUTFILE
echo $TEST_DESC | tee -a $OUTFILE
date | tee -a $OUTFILE

cd $SIMPL_HOME/bin
fclogger -n LOGGER > $TEST_DIR/results/junk &

cd $SIMPL_HOME/softwareICs/stimulator/bin
receiver -n BOBR -l LOGGER &

stimulator -n BOBS -r BOBR -l LOGGER -b &
recvTester -n BOBT -s BOBS -f test_001 -l LOGGER

echo stopping stimulator unit test
$SIMPL_HOME/bin/fcsplay BOBS
$SIMPL_HOME/bin/fcsplay BOBR
$SIMPL_HOME/bin/fcsplay LOGGER

echo "Test finished you can run posttest $TEST_NO for result"
```

As part of the stimulator SIMPL softwareIC package there is a task called **recvTester**. If you were to consult the readme file at

**\$SIMPL\_HOME/softwareICs/stimulator/test/readme**

you will find an explanation of how to run this recvTester in a manual (two console) mode.

We could integrate this into our testing framework but the two consoles will be difficult to work with. It would be better if we could combine the processes into a single console and run them all in our single runtest script.

The problem is going to be the stimulator itself. In the mode we have been running this code so far it is displaying prompts and expecting keyboard interaction. This kind of activity is all well and good if that process is run in the foreground. If however, you were to try and put the stimulator up in the background you would find it rapidly falling into an infinite loop and running amok. The reasons are complex but in simplest terms it has to do with the ill-defined STDIN file descriptor for a background process and its effect on the select() multiplexing call. The solution is very simple: prevent your background process from writing to STDOUT and your select() on STDIN will remain safely dormant.

Fortunately, our stimulator example contains a command line parameter **-b** to suppress the prompts to STDOUT and rendering it safe to be run in the background. By using this **-b** command line option, we can combine up those two consoles described in the stimulator readme into a single script that we illustrated above.

Go ahead and run the first two steps of your test by typing:

```
pretest 0001  
dotest 0001
```

At the prompt in the dotest script simply type

```
-> q
```

to quit and shutdown the test.

Before we go any further it is important to understand what just happened in this test. From that readme once again we see that recvTester is getting its *instructions* from a very simple flat file called:

```
$SIMPL_HOME/softwareICs/data/test_001
```

This file currently looks like:

```
#=====
# demo of automated stimulator testing
#=====
# Test 001
#
Test hi
Sleep 2
Test hello
```

The recvTester contains a crude parser which can:

- Discard lines which are blank.
- Discard lines which begin with #.
- Interpret all other lines as two flavours:
  - Those that begin with the letter "T" - Test
  - Those that begin with the letter "S" - Sleep

The *Sleep* line is taken as a command to do just that: put recvTester to sleep for the number of seconds specified as the second field on the line.

The *Test* line is taken to mean "**Compose and send a TEST\_MSG using the second parameter as the string**".

In other words our little test here involved sending two messages:

```
hi
hello
```

to our stimulator via its SIMPL interface. The stimulator in turn would dutifully send those on to our **receiver** the actual object of our test.

If we were to examine our trace log,

**\$TEST\_HOME/testing/test0001/results/junk**

it might look something like:

```
fclogger starting
[receiver:initialize ] 074015.205 myName=<BOBR> myslot=0
[receiver:initialize ] 074015.207 trace logger mask = 0x00FF
[receiver:receiver   ] 074015.208 starting
```

```
[stimulat:initialize ] 074015.220 myName=<BOBS> myslot=0
[stimulat:initialize ] 074015.221 trace logger mask = 0x0OFF
[recvTest:initialize ] 074015. myName=<BOBT> myslot=0
[recvTest:initialize ] 074015.236 stim name=<BOBS> stimslot=3
[recvTest:initialize ] 074015.236 full infile=<~/softwareICs/data/test_001>
[recvTest:initialize ] 074015.237 trace logger mask = 0x0OFF
[recvTest:recvTester ] 074015.237 starting
[recvTest:runttest ] 074015.238 filename=<~/softwareICs/data/test_001>
[stimulat:stimulator ] 074015.260 TEST str=<hi>
[receiver:receiver ] 074015.262 TEST str=<hi>
[receiver:receiver ] 074015.262 TEST reply str=<reply #0>
[stimulat:hndlReply ] 074015.263 TEST reply str=<reply #0>
[stimulat:stimulator ] 074017.275 TEST str=<hello>
[receiver:receiver ] 074017.276 TEST str=<hello>
[receiver:receiver ] 074017.277 TEST reply str=<reply #1>
[stimulat:hndlReply ] 074017.277 TEST reply str=<reply #1>
[recvTest:runttest ] 074017.279 rc=-1
[recvTest:recvTester ] 074026.857 done
```

Notice the **receiver** lines with the **TEST** message displayed. These are the key lines we want to see in our log if our test was successful. Note that if we were to run this junk file through a series of *greps* we could extract just these lines:

```
cat junk | grep receiver | grep "TEST str"
```

If you were to do this you should end up with:

```
[receiver:receiver ] 074015.262 TEST str=<hi>
[receiver:receiver ] 074017.276 TEST str=<hello>
```

If we were to arrange to redirect this *grep* output to a file, we could then prepare an expected result file. A simple *diff* between these two files would give us a pass/fail criterion for this test. To accomplish this we have to deal with the trace logger date stamps in our output because these will presumably change each time we run our test and ruin our *diff* strategy.

We can massage our output file through a set of filters to accomplish what we want. If we examine our trace log extract above carefully, we can decide what we want our little filter to do with an incoming line prior to printing it to stdout. The first thing to notice is that the time stamp section is all lined up on a nice fixed column on every line.

If we could simply replace the digits in this time stamp with a fixed pattern, we could be on our way to using **diff** as our pass/fail criterion. If you

count columns you will find that the time stamp is between column 25 and 34. Therefore we need a filter which when run as:

**cat junk | funfilter**

will simply replace the time stamp field (ie. col 25-34) with **xxxxxx.xxx**

Here's a version for such a filter:

```
/*=====
*
*  funfilter.c
*
*  Main source file for
*  SIMPL book test filter
*
=====*/
#include <stdio.h>

int main()
{
char line[80];

while(fgets(line, 79, stdin) != NULL)
{
    if(strlen(line) > 28)
    {
        line[24] = 'x';
        line[25] = 'x';
        line[26] = 'x';
        line[27] = 'x';
        line[28] = 'x';
        line[29] = 'x';
        line[31] = 'x';
        line[32] = 'x';
        line[33] = 'x';
    }
    printf("%s", line);
}
}
```

Go ahead and run your junk file through your newly minted funfilter and you should see the time stamp get replaced by x's. We now have all the tools to do that pass/fail part within our posttest script. Once again, for those of you familiar with the **vi** editor you can simply type:

**posttest 0001 e**

What we will need to integrate into this script are the following "stages".

- The grep sequence above to isolate the lines we really want from the trace log.
- The funfilter sequence above to replace the time stamp with x's.
- A diff comparison to see if these lines match our expected ones.

Here's a posttest script which will generate a pass/fail notification by comparing with a hand crafted expected output file at:

**\$TEST\_HOME/testing/test0001/data/grepfile.001**

```
#!/bin/bash

TEST_NO=0001
TEST_DESC="SIMPL book test1"

TEST_DIR=$TEST_HOME/testing/test$TEST_NO
OUTFILE=$TEST_DIR/results/aftershot

MYFILTER=$HOME/simplbook/src/funfilter
GREPFILE=$TEST_DIR/results/grepfile
GOODFILE=$TEST_DIR/data/grepfile.001

echo "preparing results - test#$TEST_NO, please wait" | tee $OUTFILE
date | tee -a $OUTFILE

echo "====" | tee -a $OUTFILE
cat $TEST_DIR/results/test.out | tee -a $OUTFILE

echo "====" | tee -a $OUTFILE
echo "Here is the test output" | tee -a $OUTFILE

cat $TEST_DIR/results/junk \
| grep receiver \
| grep "TEST str" \
| $MYFILTER | tee -a $OUTFILE | tee $GREPFILE

echo "====" | tee -a $OUTFILE

echo "comparing with expected results" | tee -a $OUTFILE
diff $GREPFILE $GOODFILE
DONE=$?
if [ $DONE = 0 ]
```

```
then
  echo "*** PASSED ***" | tee -a $OUTFILE
else
  echo "*** FAILED ***" | tee -a $OUTFILE
fi

echo "===== | tee -a $OUTFILE

echo "done preparing results for test#${TEST_NO}"
```

There are more examples of the SIMPL Testing Framework in both the SIMPL project tarball and the project part of this book beginning with Chapter 11.

## 9.2 Summary

We have gone into some detail in this section to illustrate the SIMPL Testing Framework (STF). We have shown how a test can be built up from basics using the public softwareICs code repository for seed code. Using the SIMPL trace logger output we have shown how with a few simple steps one can make the STF yield a pass/fail criterion.

Testability is an essential ingredient of any software application. The STF makes managing the testable SIMPL applications easier.

## 9.3 Next Chapter

In the next chapter we are going to take a look at a little known feature inherent in the SIMPL design: virtualization through multiple sandboxes.

# Chapter 10

## SIMPL Sandbox

### 10.1 Sandboxes for Testing

In the previous chapter we talked about the SIMPL Testing Framework (STF). Testable code is an important feature of SIMPL applications. One of the under-used features SIMPL is the ability to run duplicate or parallel applications in separate SIMPL sandboxes. This can help with SIMPL testability.

There are many scenarios where multiple SIMPL sandboxes might find uses. One of the obvious candidates would be to facilitate release management of a multiple SIMPL module application. When a new version of that application becomes available, it might be prudent to deploy it first in a separate SIMPL sandbox while maintaining the original deployment in the original SIMPL sandbox. That way, if issues arose that called for a temporary rollback this could easily be accommodated.

Recall from Chapter 4 on the SIMPL core library, that SIMPL *name\_locate* calls work off a special directory denoted by the FIFO\_PATH environment variable. We call that directory the *SIMPL sandbox*. Most SIMPL developers employ only one single SIMPL sandbox and export only one global FIFO\_PATH variable. However, environment variables can be easily overridden in a start up script where they only have local scope. In the examples below the SIMPL benchmark is run in two different sandboxes with the same SIMPL name in each case.

```
#!/bin/bash
# startup for sandbox 1

export FIFO_PATH=/home/sandbox1
```

```
cd $SIMPL_HOME/benchmarks/bin  
receiver -n BOBR &
```

```
#!/bin/bash  
# startup for sandbox 2  
  
export FIFO_PATH=/home/sandbox2  
  
cd $SIMPL_HOME/benchmarks/bin  
receiver -n BOBR &
```

Multiple sandboxes which are set in startup (or STF scripts) can be viewed with the same lens as the much more complex virtual machines.

The reasons for doing something like this are not unlike the reasons why someone might want to run multiple OS images in virtual machines. Principal amongst those reasons is that this mode of operation can offer flexibility on start up and shutdown of one SIMPL application without affecting the status of the others. Another reason people use virtualization is to facilitate migration to (and testing of) newer versions of the same application. Multiple SIMPL sandboxes can help in that regard also. Another characteristic of virtualization is that multiple virtual environments almost always exist on a single physical machine. This is certainly the case for multiple SIMPL sandboxes.

Virtualization is normally used to provide an isolated working environment. This is particularly true for the namespace. ie. different SIMPL sandboxes can reuse the same SIMPL names. However, occasionally it is desirable to pass a SIMPL message between a SIMPL process in one sandbox to another SIMPL process in another sandbox. An example may be when a SIMPL application undergoes a significant upgrade in business logic but not in a GUI interface. The original application including the GUI could be left running in one sandbox and the new business logic modules in a second sandbox. In order to *share* the GUI module the new business logic would need to exchange inter-sandbox messages. For this purpose tunnels are used for inter-sandbox transport in a manner not dissimilar to surrogates for network messaging.

As you may recall from Chapter 4, SIMPL exchanges messages in shared memory and synchronizes this exchange by passing the shared memory ID over a FIFO.

A tunnel is a SIMPL process that SIMPL name attaches as its counterpart in the other sandbox and connects two FIFOs together. Tunnels come in two flavours:

- Tunnels for the send exchange between the sender and the receiver.
- Tunnels for the reply exchange back to the sender.

A typical sandbox with tunnels might look like Figure 10.1.

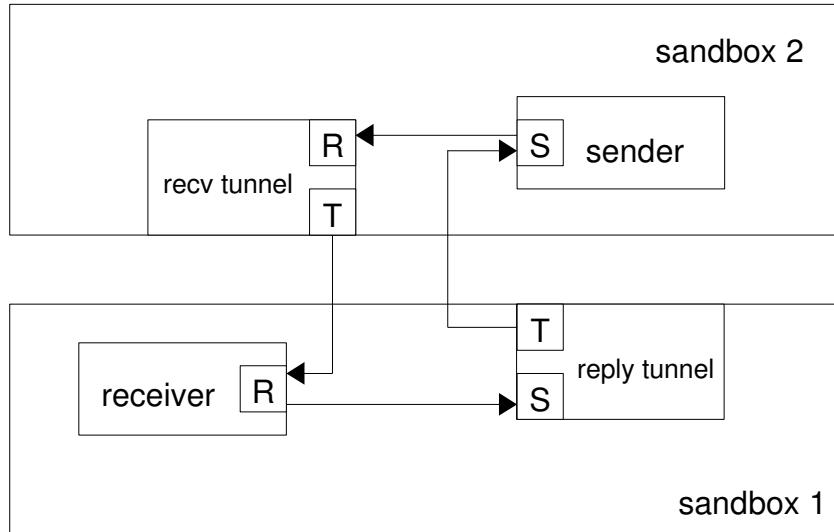


Figure 10.1: SIMPL Sandbox

We want to keep our sender and receiver code completely agnostic to the presence of multiple sandboxes and tunnels between those sandboxes. This is much the same viewpoint which was adopted to make SIMPL processes network agnostic through the use of surrogates. As such, a tunnel is really just a surrogate for inter-sandbox communication.

In fact all a tunnel has to do is to reroute the shared memory ID from one FIFO to another (the **T** box in Figure 10.1). Since the actual SIMPL message is stored in a shared memory area owned by the sender, and we are restricting tunnels to sandboxes on the same machine, the rest of the SIMPL message exchange can go as before.

Let's examine the sender->receiver message exchange illustrated in Figure 10.1. As already mentioned, the sender and the receiver are straightforward SIMPL applications with nothing special added for inter-sandbox message exchange.

The receiver tunnel process is activated on sandbox 1 to act as the *placeholder* for the actual receiver which lives in sandbox 2. In fact, this tunnel

assumes the SIMPL name of the real receiver. As such, when the sender does the ***name\_locate*** to begin the message exchange it finds what it believes to be the desired receiver local to its own sandbox. The SIMPL message exchange begins with the sender in sandbox 1 adding the bytes associated with the message into its own block of shared memory. Next, the sender pushes the shared memory ID for this message block onto the FIFO which connects the sender with what it believes to be the receiver, but what is actually the receiver tunnel process.

Meanwhile, when the receiver tunnel is activated it establishes a second connection to the receive FIFO associated with its real receiver in sandbox 2. When the shared memory ID arrives in the receiver tunnel's SIMPL FIFO it is simply copied over to this sandbox 2 FIFO and the receiver tunnel's job is done. The real receiver now sees the shared memory ID arrive in its FIFO and reacts as if a normal SIMPL message has arrived. ie. opens the shared memory segment and accesses the data.

Now, a second tunnel process was started along with the SIMPL receiver in sandbox 2: the reply tunnel. This reply tunnel assumes the SIMPL name of the sender in the sandbox 1 and upon start up opens a second channel back to the sender's reply FIFO.

Once the receiver processes the message it adds the response data back into the shared memory segment and pushes a return code onto the sender's reply FIFO which it believes is the reply tunnel process (because it has the same SIMPL name as the sender). Once the reply tunnel process sees this return code it simply copies it over to the real reply FIFO and its job is complete.

The sender then sees the response as if it came from the receiver. We have achieved our objective of a totally transparent SIMPL message pass between two SIMPL sandboxes.

## 10.2 Summary

Virtualization has become popular because it is useful. Virtualization of SIMPL via multiple SIMPL sandboxes will become popular because it too is useful.

## 10.3 Next Chapter

In the next section we are going to explore a fictitious SIMPL project with the aim of illustrating the SIMPL approach to application building in a more concrete manner.

# Part IV

# SIMPL Example Project



# Chapter 11

## A SIMPL Project - Putting It All Together

The aim of this part of the book is to explore a straightforward example project. After all, most of our readers will also be programmers and will want to see a software tool like SIMPL in action. They will want to see the project source code and even modify it as they explore and read along. As such we want to choose an example project which includes as many aspects of SIMPL as practical yet doesn't take up too much space with unnecessary details. Our project, an Internet distributed collection of word and number puzzle solvers, would not likely be found in the business world. However, we'll use it to illustrate many of the principles discussed in the book so far and perhaps even more importantly we'll show that SIMPL programming can be fun.

### 11.1 Project Specification

The project we have in mind is admittedly contrived. Hopefully it is complex enough to show as many facets of the SIMPL approach as possible, but not be so complex as to lose the story thread.

The specification (or spec for short) for our collection of word and number puzzle solvers calls for both the solver engines and the puzzle submitting clients to be connected via the Internet. The engines will be hosted on servers. The users will access those puzzle engines from any of the popular desktop OSs.

This is a somewhat daunting spec at first glance. We are going to illustrate how the SIMPL approach involves translating this spec into a SIMPL system picture. Armed with this full system picture we will chose to implement the

simplest subsystem first.

## 11.2 The Design Meeting

Typically the designers, armed with the spec, would convene a meeting where various implementations would be discussed and debated on a white board. The outcome of such a meeting would be a system picture something like Figure 11.1.

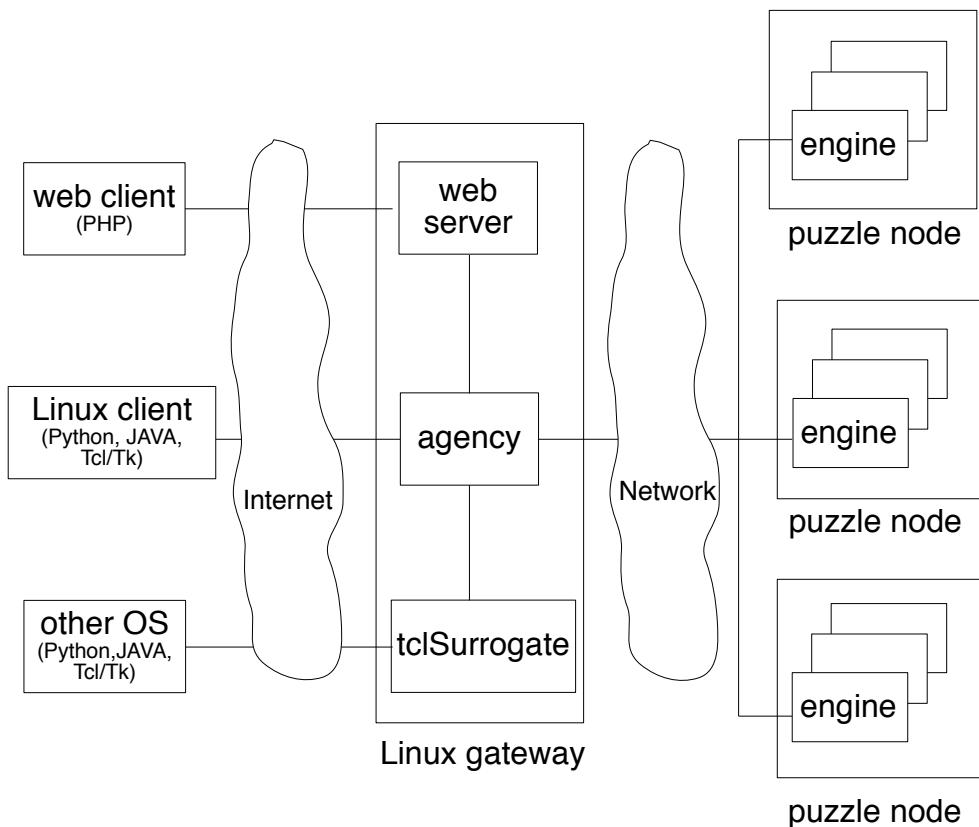


Figure 11.1: System Picture

On the left hand side the various client interfaces are illustrated. On the right hand side the various puzzle solver engines are illustrated. In the middle is the central Linux gateway which connects the two. For Linux desktops it will be possible to submit puzzles using a Java/Swing, Python/Tk, or a Tcl/Tk client implementation. For the other desktop OSs such as Windows or Mac, a Java/Swing, Python/Tk, or Tcl/Tk client implementation will also be available.

All of the desktops will support web applet access via the Tcl/Tk Firefox plug-in as well as a PHP based web interface using Python CGI.

The puzzle solver engine nodes will be restricted to Linux OS servers. The engines themselves could be implemented in any number of programming languages but we'll restrict ourselves to 'C'. 'C' is a great language for implementing SIMPL modules and it is powerful enough for our puzzle solving algorithms. To handle the simultaneous client load the engines would be implemented as forking *agents* (recall Chapter 6).

The gateway node will be configured to host a web server and the SIMPL TCP/IP surrogates as well as the tclSurrogate. In addition, the gateway node will host the central puzzle solver module derived from the agency softwareIC to allow multiple concurrent Sudoku sessions. Recall Section 6.8 in Chapter 6. The Linux clients will use the TCP/IP surrogate pathway to establish SIMPL connectivity with the puzzle engines. The other desktop OS clients and the browser applets will use the tclSurrogate interface to establish SIMPL connectivity. The gateway web server will host the PHP interface into the system.

The system picture has certainly helped solidify our spec, but it's still complex. Next up, the design team needs to come up with a work plan.

## 11.3 The First Cut

The SIMPL approach closely follows Kent Beck's Extreme Programming<sup>1</sup> when it comes to choosing the initial implementation. We'll pick the simplest possible subsystem which has any business value and implement that first. We'll do this with full confidence that this base system can be evolved in a systematic manner to arrive at the final feature set described in the spec.

This approach has several advantages. Firstly, it gets the team energized and productive in the quickest possible way. Secondly, it gives the users some tangible progress in the form of usable code at the earliest date. This lowers both the ultimate cost and risk associated with any software project. In addition, it provides the project team with the adaptability to adjust to the invariably changing spec.

How do we simplify our system picture to this base implementation? The spec calls for multiple puzzle engines. We'll settle on Sudoku as our first engine implementation. Sudoku has a tidy set of puzzle rules and a straightforward interface so it makes a good candidate. The spec calls for several different client

---

<sup>1</sup>Extreme Programming Explained by Kent Beck, Addison-Wesley Pub., 2000, ISBN 0-201-61641-6

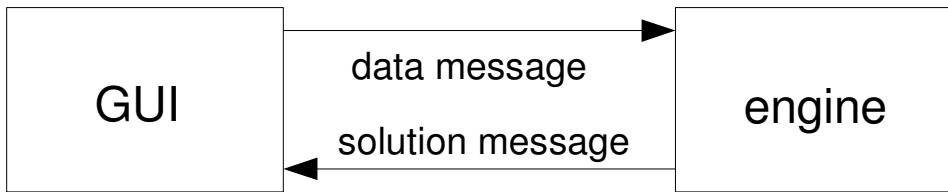


Figure 11.2: Sudoku Subsystem Picture

implementations. We'll settle on Python as the first go. Why Python? We could have chosen to create the GUI using 'C'/X Windows, Java or Tcl/Tk. Python is a popular object oriented scripting language and would likely have a shorter development time than the other choices. The spec calls for the engine to be a forking agent capable of handling multiple simultaneous clients. We don't need this level of sophistication on the first cut, so we'll settle on an engine which is a primitive SIMPL receiver and a GUI which is a primitive SIMPL sender. (see Figure 11.2).

We'll plan to evolve this base implementation through a series of straightforward transformations into our full feature set. For example, the planned transformation sequence for the engine might look like the following:

- Convert to a forking receiver.
- Convert from a forking receiver to a forking agent.

Similarly the initial Python client software could evolve as:

- Port the Python client to a Tcl/Tk Linux client.
- Port the Python client to a Java/Swing client.
- Port the Tcl/Tk Linux client to a tclSurrogate capable client.
- Port the tclSurrogate client to a web applet.
- Develop a PHP interface which duplicates the functionality above.

## 11.4 Sudoku

So initially we'll be creating programs that solve sudoku puzzles.<sup>2</sup> To start with let's take a look at what makes up a sudoku puzzle. For us it will consist of a 9x9 or a 16x16 matrix of numbers. In the first case the numbers are the decimals 1-9. In the second case the numbers are the hexadecimals 0-F. When the puzzle is given, only some of the numbers in the matrix are non-blank. A solution to the puzzle involves filling in the blanks such that all rows contain all available numbers with no repetition, all columns contain all available numbers with no repetition and all submatrices, 3x3 and 4x4 respectively also contain all available numbers with no repetition. See Figure 11.3 for an example of typical 9x9 puzzle. The submatrices in the figure appear as shaded or not shaded.

	1	9						
		8		3		5		
7		6				8		
		1		6	8		9	
8			4					7
9	4					1		
				2				
			8		5	6	1	
	3	7				9		

Figure 11.3: A Typical 9x9 Sudoku Puzzle

What would be our approach to this task? Our first cut meeting above proposed an initial solution which consists of a Python GUI to display the puzzle and also some way to accept the correct numerical values. As well, we proposed a ‘C’ program that solves the puzzles based on the given values. The Python GUI will be a SIMPL sender and the Sudoku engine will be a SIMPL receiver. We are going write one GUI program in Python/Tk and one engine program in ‘C’. The end user will use this GUI for collecting input data, sending that data to the engine program and then displaying the solution. The engine program will accept a puzzle and contain an algorithm for solving it.

---

<sup>2</sup>If you wish to follow along with some real code in front of you the entire Sudoku project code is available at <http://www.icanprogram.com/simplBook>.

Notice what has happened here. The modular nature of SIMPL has allowed us not only to simplify our problem into a very basic initial implementation; it has also allowed us to chose programming languages appropriate to each portion of that implementation. Once we can agree on the structure of the SIMPL messages which will be exchanged between these two modules, we are in a position to develop them in a parallel fashion. It appears that we have broken up the task functionality in a way that exploits a natural division of labour. Moreover, we will run the GUI and engine programs on one host, thus eliminating the need for any surrogates and removing any possible problems due to networking issues.

## **11.5 Summary**

At this point the project team has an overall picture. It also has a near term work plan, albeit at a much simpler level than the overall picture. We are going to develop a Sudoku puzzle solver first. We are going to create a Python GUI sender which will exchange messages with a Sudoku puzzle engine which is a SIMPL receiver written in ‘C’.

## **11.6 Next Chapter**

In the next chapter we are going to show how the project team begins the actual work by setting up the SIMPL Testing Framework (STF) associated with the Sudoku puzzle solver task.

# Chapter 12

## Test Plan

### 12.1 Sudoku Test Plan

At this point the project team has white-boarded the project picture. This project picture has then been simplified into the first piece of work: the Sudoku puzzle solver. The next task for the project team will be to design the SIMPL Testing Framework (STF) tests.

As always there will be unit tests designed to exercise particular SIMPL modules in the Sudoku puzzle solver system. There will also be integration tests whose purpose is to exercise the collective system of all the modules.

The test plan will mirror the project plan in that it will begin with a Python GUI and a ‘C’ Sudoku solver engine and evolve to include Tcl and Java GUIs as well as a Python CGI interface.

In the first phase (Python GUI, ‘C’ engine) the unit tests might look like:

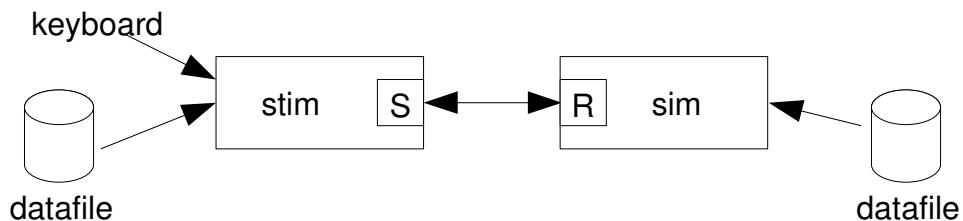


Figure 12.1: sb000 - sim/stim unit test

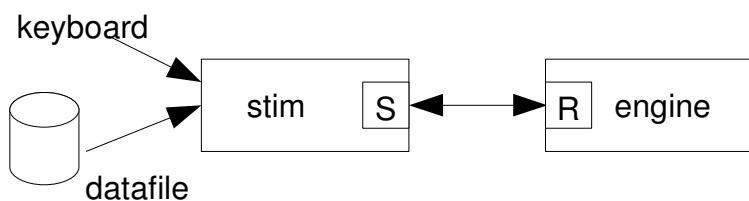


Figure 12.2: sb001 - Sudoku engine unit test

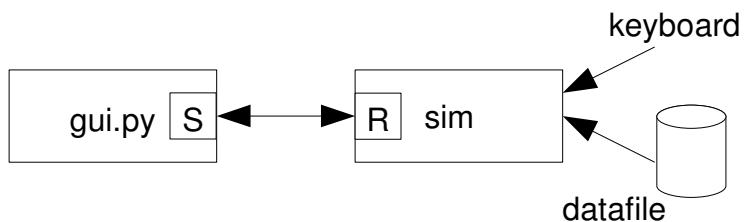


Figure 12.3: sb002 - Python GUI unit test

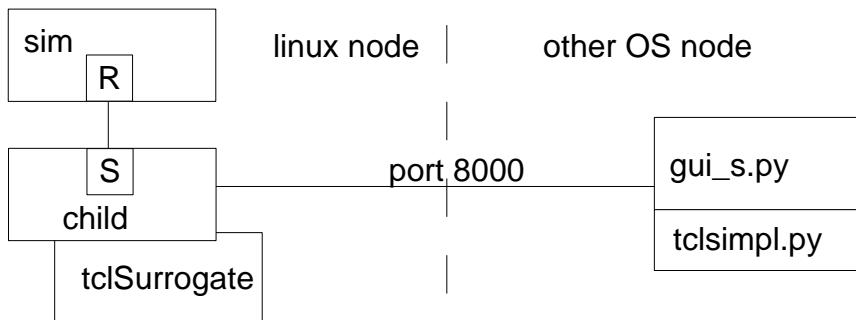


Figure 12.4: sb002s - Non-Linux Python GUI unit test

These tests are designed to exercise the two SIMPL modules in the first phase: Python GUI and ‘C’ engine as well as the actual test stubs themselves.

The integration test for this phase might look like:

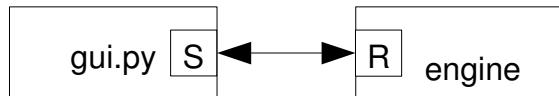


Figure 12.5: sb004 - Python GUI/Sudoku engine test

## 12.2 The Unit Tests

The SIMPL toolkit promotes modular problem decomposition. SIMPL unit tests are designed to exercise one particular SIMPL module thoroughly. The SIMPL messages originate with, or are absorbed by test stubs in the unit tests, depending on whether the SIMPL module is a receiver or a sender. The purpose for a unit test is to exercise as many pathways through the coded algorithm of the test subject as possible.

In this first phase of the project we have a two SIMPL module application to unit test. The Python GUI module is unit tested against an engine simulator. The Sudoku solver engine module is unit tested with the help of a stimulator. By virtue of this design, GUI testing team and the puzzle algorithm testing team can completely decouple from one another.

Take the engine unit test (sb001) for example. We'd want at a minimum to submit a valid, invalid and unsolvable puzzle to test the engine's ability to handle those cases. We'd pay particular attention to the message content being submitted and the message content being replied back. In some projects it may pay to invest the time to automate these unit tests to allow them to be run regularly from a *cron* script and to have them yield a pass/fail criterion. Our example project is simple enough that we'll stick with straightforward hand-run and hand-examined unit tests.

The Python GUI unit test comes in two flavours because the Python GUI can be run in two distinct modes depending on whether the code is destined for deployment on a Linux node (sb002) or a non-Linux node (sb002s). Despite the difference in the execution environment the Python GUI code will be largely the same with the only difference being in the SIMPL library being invoked (csimpl.py or tclsimpl.py).

## 12.3 The Integration Tests

Integration tests on the other hand, are designed to exercise combinations of SIMPL modules which form the full SIMPL application. Integration tests don't often employ test stubs except perhaps at the boundaries where they substitute for real world I/O. In this project the integration test (sb004) will not employ any test stubs. The integration test is employed to test the ability of the entire SIMPL application to work correctly together. This is where one would look for timing issues or GUI usability issues under simulated real world loads.

In basic applications like this one, once the engine module has passed the sb001 unit test, the real engine module can be used to exercise the follow on GUI teams efforts (sb005 for the Tcl/Tk team, sb006 for the Python web team and sb007 for the Java team). If problems develop that need more refined control on the engine side, it is quick change for these teams to swap in the engine simulator into their tests.

## 12.4 Summary

The project team now has a near term test plan. All the tests have been identified and *white-boarded*.

## 12.5 Next Chapter

In the next chapter we are going to show how the project team begins the actual work by setting up the project directory tree and begins to populate it with seed code that can allow immediate execution of the STF test plan scripts.



# Chapter 13

## Directory Tree

### 13.1 Good Housekeeping

At this point in the project, we have designed our unit and integration tests for the *simplest possible subimplementation* of our puzzle solver system.

The next step involves making housekeeping decisions on the source code and testing directory tree. Once the tree is designed it will be populated with suitable seed code from the softwareICs (or another suitable) SIMPL code repository.

In our specific case we have settled on a simplified sender-receiver pair for the Sudoku puzzle solver portion of our system. We have designed unit tests for the various configurations of this pair plus an additional proof test for the stim and sim test stubs themselves.

We have elected to name our project **simplBook**. This will become the apex directory in our tree. To aid in location of this tree on different virtual developer trees, we will designate an environment variable which points to the apex directory. We will call this environment variable: *SIMPLBOOK\_HOME*.

Inside this *SIMPLBOOK\_HOME* directory we will create our top level Makefile for the project. When executed, this top level Makefile will build everything associated with the puzzle solver project. A good example of such a top level Makefile is the one associated with the SIMPL toolkit itself. Also residing in this apex directory will be the module subdirectories. In our case we will begin with a subdirectory called *sudoku*. The top level include subdirectory will exist at this level as well. It will be populated with any global headers required by all the modules in the project. Another directory at this level will be called *testing*. This will be home to all the test scripts and data files associated with the SIMPL testing framework test plan described in Chapter 12. Finally, there

will be a *bin* directory where all submodule executables will be promoted on a successful *make install*.

At this point our directory tree looks like:

```
/simplBook/Makefile  
/simplBook/include/  
/simplBook/bin/  
/simplBook/sudoku/  
/simplBook/testing/
```

The sudoku subdirectory itself will contain a Makefile. This Makefile will be called from the top level Makefile and will be responsible for building all the elements of the sudoku module. Also residing at this level in the tree will be three subdirectories associated with the source code of the sudoku module: *src*, *include*, *test*. The *src* subdirectory will contain the sudoku engine and client source code and the *include* subdirectory will contain any local header files associated with the sudoku module. The *test* subdirectory will be home to all the sudoku test stubs. Finally, there will be an intermediate build area for executables associated with the sudoku module in a directory called *bin*. A simple make at the sudoku level will populate this *bin* subdirectory. Unit test scripts would point into this *bin* directory for the executables to run.

At this point the sudoku subtree would look like:

```
/simplBook/sudoku/Makefile  
/simplBook/sudoku/bin/  
/simplBook/sudoku/src/  
/simplBook/sudoku/include/  
/simplBook/sudoku/test/
```

## 13.2 Seeding the Code

At this stage we have only populated our tree with a couple of very generic high level Makefiles. It is time to seed the tree with some source code and source code Makefiles so that we can begin the construction of operational test scripts in our framework.

A good place to go for seed code is our softwareICs code repository. Since our unit test design calls for a stimulator a great place to start might be with

the stimulator softwareIC source code.<sup>1</sup> We'll need to rearrange things a little because our sudoku/src is going to contain the sudoku engine, a SIMPL receiver. We'll want to move a copy of the receiver.c and receiverInit.c into the sudoku/src directory and rename those as engine. We'll want to move the simulator code into the sudoku/test subdirectory and rearrange the Makefiles to allow this seed code to compile. The copy of receiver left in the test subdirectory is destined to become the simulator in our unit tests.

At this stage our **sudoku/src** portion of the tree looks something like:

```
/Makefile  
/engine.c (copy of receiver.c)  
/engineInit.c (copy of receiverInit.c)
```

and **sudoku/test** looks something like:

```
/Makefile  
/stimulator.c  
/stimulatorInit.c  
/stimulatorUtils.c  
/sim.c (copied from receiver.c and renamed sim.c)  
/simInit.c (copied from receiverInit.c and renamed simInit.c)
```

### 13.3 Seeding the Tests

If we examine each of these tests we quickly observe that they all are very similar to the SIMPL unit test for softwareICs stimulator (s0008). As such we can use the *copytest* wrapper script<sup>2</sup> to clone that SIMPL test into each of our tests.

**copytest s0008 sb000**

With a quick edit for each description and each test script suite to insert the actual test number in place of the SIMPL test number we have operational tests in our test plan. They don't do what we want as yet, but that is the subject of the next chapter.

---

<sup>1</sup>If you wish to follow along with some code in front of you, we have included a representative body of seed code in the main simplBook sudoku tarball under the **sudoku.seed** subdirectory. The process of obtaining and installing that tarball is described at <http://www.icanprogram.com/simplBook>.

<sup>2</sup>The SIMPL Testing Framework will install a number of wrapper scripts under **SIMPL\_HOME/scripts**, of which *copytest* is one.

Once we get the whole show recompiling we will want to change the runtest script associated with test sb000 (the sim-stim test) to pickup its executables from our new build area. At this point test sb000 can be run, albeit with the same result as when it ran as a SIMPL unit test clone.

## 13.4 Summary

In this chapter we have designed our entire directory tree for the first phase of this project. We have populated that tree with seed code from the SIMPL softwareICs repository and the SIMPL STF tests. After renaming some files and rearranging some Makefiles we can get the whole thing compiling and our main unit test can be executed.

## 13.5 Next Chapter

We are now ready to begin the actual Sudoku design process. In the next chapter we are going to begin by designing the tokenized messages we are going to require for our Sudoku puzzle solver and we are going to begin to show how the header files for our project will be renamed and populated with our new message structures.

# Chapter 14

## Sims and Stims

”A ‘C’ programmer only writes one program, then beats it up for evermore.”  
—Anonymous

### 14.1 Unit Testing

At this point we have various unit tests defined. We have populated these unit tests with seed code from the SIMPL softwareICs repository. Everything compiles and executes, only it doesn’t contain any Sudoku code as yet.

We have talked previously about the extendability of SIMPL. This is usually taken to infer that SIMPL designs readily allow simpler things to be transformed into more complex forms. In the real world of programming this is often true but it is not always the case. It is also possible to take more complicated systems and rework them into simpler systems because they were originally designed with SIMPL in mind. In this section we are going to do just that, take the relevant softwareICs as a starting point and simplify them into the sort of code that we need to fulfill our teaching obligations to the reader and obtain code snips that are amenable to the book. We should mention that SIMPL designs also provide convenient lateral modifications. ie. from one system to another of similar complexity. This would be the case when an existent system has the sort of design and message passing requirements needed for a proposed system: Why reinvent the wheel?

Now that we have seed code compiling, our method for migrating this code to the final form will be essentially iterative following these steps:

**Step 1:** Make changes to seed code.

**Step 2:** Get the code recompiling.

**Step 3:** Run the unit test to see results in trace log.

**Step 4:** Go back to Step 1.

The advantage of starting with a fully trace log enabled body of seed code and an operational unit test cannot be overstated. This allows the Sudoku developer team to make methodical and incremental changes to the code all the while ensuring that it recompiles and executes at each stage. Problems are quickly discovered and easily isolated and repaired using this approach to program development.

## 14.2 What sort of Messages Do We Need?

Our seed code contains tokenized SIMPL messages which are exchanged between the stimulator and the engine (or sim). However, at this stage these tokenized message definitions are still those from the seed code in the repository. The Sudoku developer team will now need to begin the SIMPL code design process by defining the tokens and message structures that will be required to operate the Sudoku puzzle engine.

The first message that we need to define pertains to the information gathered by the GUI and then sent to the engine for processing. The available information is:

1. The size of puzzle to be solved, 9x9 or 16x16.
2. The values of the given matrix elements. That is 1-9 or 0-F and a blank for an unknown value.

The size of the puzzle could be relayed by the message token. The body of the message could be either  $9 \times 9 = 81$  or  $16 \times 16 = 256$  characters with the given matrix elements set to the appropriate numbers and the unknown values set to spaces. This is all the information that the engine program requires in order to attempt a solution.

The second message that we need to define is the reply message to the GUI from the engine. What does the GUI require? It simply needs to know whether the puzzle could be solved and if so, what the solution is. Again, we could use the token value of the message to indicate success or failure. The body of the message could be the solution.

From a ‘C’ code perspective, the sent and reply messages would both look like the following:

engineMsgs.h

```
typedef enum
{
    DO_9,
    DO_16,
    PUZZLE_FAILURE,
    ERROR,
    BAD_TOKEN,
    MAX_ENGINE_TOKENS
} ENGINE_TOKEN;

typedef struct
{
    int token;
    char elements[256]; // maximum possible size
} SUDOKU_MSG;
```

We have defined a message structure of type SUDOKU\_MSG. The GUI can set the structure member called token to define the puzzle size and the structure member character array called elements can take up to the largest size of puzzle. The engine can use the same structure by setting the token to:

1. DO\_9/DO\_16 in the case of a successful solution.
2. PUZZLE\_FAILURE in the case of no solution.
3. ERROR if a problem of some sort is encountered.
4. BAD\_TOKEN if the received token is unknown.

The engine will return the solution in the elements array in the case of a successful solution. Note that this situation is somewhat unusual in that the structures of the sent and replied messages are usually very different. Nevertheless, the simplicity of the situation is gratifying. We have now defined the interaction between the GUI and the engine.

Now that we have the tokenized messages designed it is time to add that to our main Sudoku engine header file. In our case that header is now called *receiverMsgs.h*. We will begin by changing the name to *engineMsgs.h* and proceed to add our tokens and tokenized message structures to that file. We’ll have

to change all occurrences of `receiverMsgs.h` in our seed code to `engineMsgs.h` to get things safely recompiling once again. Recall that `receiverMsgs.h` is a general type of header file like the `softwareICs` that we are making specific to our task.

## 14.3 Global Variables

The global variables we will be specifying in our engine and simulator currently share a common header called `receiver.h`. We need to rectify that now by cloning that header once and renaming the pair as `engine.h` and `sim.h`. The same is true of the header containing function prototypes: `receiverProto.h` which gets cloned as `engineProto.h` and `simProto.h`. Once again all instances of these previous header names are simply replaced by the new ones and all of the code should recompile and run once again.

Each of the SIMPL modules (stimulator, engine and simulator) are composed of multiple source files. We have adopted the **ALLOC** scheme for sorting out whether those global variables are declared outright (as in the source file containing `main()`) or as **extern** (as in `Init` and `Utils` files).

## 14.4 Simulator and Stimulator

At this point we still have identical seed code for both our simulator and our engine. It makes sense at this stage to focus on the simulator recognizing that once we get that operational, the engine can absorb those changes and evolve from there. Furthermore, our simulator can then be added to the GUI unit tests (`sb002`, `sb003`, and `sb003s`, see Figures 12.3, 18.3, and 18.4) as the engine substitute.

Recall that a simulator is a stub program whose only purpose is to receive a canned message from a sender and then reply to that sender in order to aid in testing the SIMPL aspect of the sender code, ie. the messaging scheme. It processes as little as possible because its main purpose is to exercise the SIMPL interface. In this way a sender program can be developed independently of a receiver program. For our example, we won't need the engine in order to receive a message from the GUI. The sim program will simply consist of a program that receives and replies a canned message of type `SUDOKU_MSG`. We will write the sim in 'C' so that we can use the structure definition as it stands. As well, when we are satisfied with the efficacy of the sim we can use it as a solid starting point for the engine. The essential difference between sim and engine will be that the later contains the complete Sudoku processing algorithm.

We also want to develop a stub for the GUI program. Fortunately, we have inherited a very capable stimulator when we seeded our code from the SIMPL repository. Recall that a simulator is a small program that sends a canned message to a receiver and then obtains a canned reply from that receiver. In this way a receiver program can be developed and tested independently of any sender program. For our example, the simulator program will simply consist of a program that sends a canned message of type SUDOKU\_MSG to the engine and then obtains a reply either indicating failure or success and a solution. The simulator seed code is written in ‘C’ whereas our first GUI will be written in Python. Nevertheless, the simulator will be able to ascertain the veracity of the SIMPL communications to the SIMPL receivers in all of our unit tests. Later we can take the essence of the simulator and base the actual GUI programs upon this.<sup>1</sup>

## 14.5 Simulator Logic

As has already been stated, the purpose of the simulator is to act as a SIMPL stub for the receiver we have called the engine. Let’s take a look at the code for the sim and examine the salient points. We will start with sim.h and simProto.h, header files containing definitions peculiar to sim. Note that the message and token definitions used by the sim program are the same as defined earlier as SUDOKU\_MSG and ENGINE\_TOKEN respectively.

### sim.h

```

1 // defined macros
2 #define SIZE_9          9
3 #define SIZE_16         16
4 #define SUCCESS        0
5 #define FAIL           -1
6 #define MAX_MSG_SIZE 1024
7
8 // for the logger
9 #define SIM_MARK      0x00000001
10 #define SIM_FUNC_IO   0x00000002
11 #define SIM_MISC      0x00000010
12
13 // global variables
14 _ALLOC char datapath[80];

```

---

<sup>1</sup>If you wish to follow along with some real code in front of you the entire Sudoku project code is available at <http://www.icanprogram.com/simplBook>.

```

15 _ALLOC int size; // size of the incoming puzzle (9x9 or 16x16)
16 _ALLOC char array[SIZE_16 * SIZE_16]; // maximum size
17 _ALLOC unsigned int globalMask; // for the logger
18
19 // for the incoming and outgoing messages
20 _ALLOC char inArea[MAX_MSG_SIZE];
21 _ALLOC char outArea[MAX_MSG_SIZE];

```

lines 2-6 Definitions used by sim code.

lines 9-11 Definitions used in trace logger function calls.

lines 14-17 Global variable declarations.

lines 20-21 Global incoming and outgoing message buffer declarations.

### simProto.h

```

1 void initialize(int , char **);
2 void myUsage(void );
3 int process(SUDOKU_MSG * , SUDOKU_MSG * );
4 void readData(char * );

```

lines 1-4 Function prototype declarations.

line 1 All program initializations are carried out in this function at startup.

line 2 Program usage in the case of an incorrect command line.

line 3 The function that carries out the actual program processing.

line 4 The function that reads the appropriate data file.

### sim.c

```

1 // standard headers
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <time.h>
6 #include <string.h>
7
8 // sim program headers

```

```
9 #define _ALLOC
10 #include "sim.h"
11 #undef _ALLOC
12 #include "engineMsgs.h"
13 #include "simProto.h"
14
15 // SIMPL headers
16 #include "simpl.h"
17
18 // for the logger
19 #define _ALLOC
20 #include "loggerVars.h"
21 #undef _ALLOC
22 #include "loggerProto.h"
23
24 /*=====
25     sim - entry point
26 =====*/
27 int main(int argc, char **argv, char **envp)
28 {
29     char *fn = "sim";
30     int x_it = 0;
31     int nbytes;
32     char *sender;
33     SUDOKU_MSG *in = (SUDOKU_MSG *)inArea;
34     SUDOKU_MSG *out = (SUDOKU_MSG *)outArea;
35     int msgSize;
36     int ret;
37
38 // initialize variables, ready program for SIMPL communications etc.
39 initialize(argc, argv);
40
41 // log the startup
42 fcLogx(FILE, fn, globalMask, SIMMARK, "starting");
43
44 while (!x_it) // loop indefinitely receiving & replying to messages
45 {
46     // receive incoming messages
47     nbytes = Receive(&sender, inArea, MAX_MSG_SIZE);
48     if (nbytes == -1)
49     {
50         fcLogx(FILE, fn, SIMMARK, SIMMARK,
51             "Receive error-%s", whatsMyError());
52         continue;
53     }
54 }
```

```
55 // decide course of action based on value of the message token
56 switch (in->token)
57 {
58     // a 9x9 puzzle
59     case DO_9:
60         // set necessary variables
61         size = SIZE_9;
62
63         // try to match the puzzle and compose reply message
64         ret = process(in, out);
65         if (ret != -1)
66         {
67             // success
68             out->token = DO_9;
69             // use msgSize NOT sizeof(MSG), C may pad out structure
70             msgSize = sizeof(int) + size * size * sizeof(char);
71         }
72     else
73     {
74         // failure, no solution
75         out->token = PUZZLE_FAILURE;
76         msgSize = sizeof(int);
77     }
78
79     // reply results to sender
80     if (Reply(sender, out, msgSize) == -1)
81     {
82         fcLogx(_FILE_, fn, SIM_MARK, SIM_MARK,
83             "Reply error-%s", whatsMyError());
84     }
85     break;
86
87     // a 16x16 puzzle
88     case DO_16:
89         // set necessary variables
90         size = SIZE_16;
91
92         // try to match the puzzle and compose reply message
93         ret = process(in, out);
94         if (ret != -1)
95         {
96             // success
97             out->token = DO_16;
98             // use msgSize NOT sizeof(MSG), C may pad out structure
99             msgSize = sizeof(int) + size * size * sizeof(char);
100        }
```

```

101     else
102     {
103         // failure , no solution
104         out->token = PUZZLE_FAILURE;
105         msgSize = sizeof(int);
106     }
107
108     // reply results to sender
109     if (Reply(sender, out, msgSize) == -1)
110     {
111         fcLogx(FILE, fn, SIM_MARK, SIM_MARK,
112             "Reply error=%s", whatsMyError());
113     }
114     break;
115
116     // unknown message token
117     default:
118         // reply error condition to sender
119         out->token = ERROR;
120         msgSize = sizeof(int);
121         if (Reply(sender, out, msgSize) == -1)
122         {
123             fcLogx(FILE, fn, SIM_MARK, SIM_MARK,
124                 "Reply error=%s", whatsMyError());
125         }
126         fcLogx(FILE, fn, SIM_MARK, SIM_MARK,
127             "unknown token=%d", in->token);
128         break;
129     }
130 } // end of while loop
131
132 // program is finished
133 fcLogx(FILE, fn, globalMask, SIMMARK, "done");
134
135 // remove SIMPL attachments
136 name_detach();
137
138 // exit gracefully
139 exit(0);
140

```

lines 2-6 Required standard ‘C’ headers.

lines 9-13 Headers described above. Note the use of \_ALLOC in determining the extern status of global variables used in multiple source code files.

line 16 Required SIMPL definitions.

lines 19-22 Required headers for use with trace logger function calls.

lines 29-36 Local to main() variable declarations.

line 39 All required initialization procedures are performed here including the all-important SIMPL **name\_attach**. The readies the program for SIMPL communications regardless of whether the program sends, receives or both. This initialize() function source is contained in the simInit.c file.

line 42 Inform the trace logger that the sim is starting.

lines 44-130 An infinite loop indicating that messages are to be received, processed and replied to indefinitely.

lines 47-53 The **Receive** blocks until a message arrives. The sender's SIMPL identity is pointed to by *sender*, the message is placed into the memory pointed to by *inArea* and the message cannot be any larger than the memory area indicated by *MAX\_MSG\_SIZE*. Note the error checking.

lines 56-128 We look at the value of the incoming message token in order to decide on the correct course of action. Note the switch default which catches errors in the token value.

lines 64,93 A solution to the incoming puzzle is attempted via the process() function call.

lines 80,109,121 The reply message is sent to the sender program. The message contents are set based on the findings of process() function or due to a bad message token.

The error handling in a stim or a sim is crucial because the point of sims and stims is to test the veracity of the SIMPL communications involved. The balance of the important sim code is as follows:

```
1 int process(SUDOKU_MSG *in, SUDOKU_MSG *out)
2 {
3 // datapath is global, set in initialize()
4 // size is global, set in main()
5 // array is global, set in readData()
6 int ret = FAIL;
7 char file[200];
8
```

```

9 // make the data file path and name
10 sprintf(file , "%s/valid_%d.dat" , datapath , size );
11
12 // read in the required data file into the global array
13 readData( file );
14
15 // is the given puzzle the same as that sent in by stim?
16 if (!memcmp(array , in->elements , size * size ))
17 {
18     // make the data file path and name
19     sprintf(file , "%s/answer_%d.dat" , datapath , size );
20     // set the canned answer
21     readData( file );
22     memcpy(out->elements , array , size * size );
23     ret = SUCCESS;
24 }
25
26 return( ret );
27 }
```

line 10 The path and file name containing the corresponding message element array data is set; it contains either the 9x9 or 16x16 array of canned values.

line 13 The data file of canned values is read into memory.

lines 16-24 The canned file data is compared to the incoming array of canned data and if it compares favourably, then the *canned* solution is read from another data file and loaded into the reply message.

Note that there is no actual processing in the sim, that is not its purpose. We will be able to send a predefined canned message to the sim from either the GUI or the stim and test that the message data arrives in an uncorrupted state. In fact, we will be able to send predefined failures as well in order to check error handling logic.

## 14.6 Stimulator Logic

As discussed earlier, the purpose of the stimulator is to act as a SIMPL stub for the GUI (a SIMPL sender in our first cut design). In many ways, the stim looks like a mirror reflection of its SIMPL partner the sim. We will first use our stimulator to unit test the sim and eventually the engine. Let's take a look at the code for the stim starting with its headers followed by the code body.

stim.h

```

1 // defined macros
2 #define SIZE_9          9
3 #define SIZE_16         16
4 #define SUCCESS        0
5 #define FAIL           -1
6 #define MAX_MSG_SIZE   1024
7
8 // for the logger
9 #define STIM_MARK      0x00000001
10 #define STIM_FUNC_IO   0x00000002
11 #define STIM_MISC      0x00000010
12
13 // global variables
14 _ALLOC char myName[20]; // SIMPL name
15 _ALLOC int recvID; // SIMPL ID
16 _ALLOC int size; // size of the incoming puzzle (9x9 or 16x16)
17 _ALLOC int type; // anticipated from receiver
18 _ALLOC char array[SIZE_16 * SIZE_16 + 1]; // maximum size
19 _ALLOC unsigned int globalMask; // for the logger
20 _ALLOC int backgroundMode; // to enable keyboard access
21 _ALLOC char datapath[80];
22 _ALLOC int fd;
23 _ALLOC int maxfd;
24 _ALLOC int my_fds[1];
25 _ALLOC fd_set watchset;
26 _ALLOC fd_set inset;
27
28 // for the incoming and outgoing messages
29 _ALLOC char inArea[MAX_MSG_SIZE];
30 _ALLOC char outArea[MAX_MSG_SIZE];

```

lines 2-6 Definitions used by sim code.

lines 9-11 Definitions used in trace logger function calls.

lines 14-26 Global variable declarations.

lines 29-30 Global incoming and outgoing message buffer declarations.

stimProto.h

```

1 void initialize(int, char **);
2 void myUsage(void);

```

```
3 char *skipOverWhiteSpace(char *);  
4 void makeMsg(char *, SUDOKU_MSG *);  
5 void checkMsg(SUDOKU_MSG *);  
6 void readData(char *);  
7 void printArray(void);
```

lines 1-7 Function prototype declarations.

line 1 All program initializations are carried out in this function at startup.

line 2 Program usage in the case of an incorrect command line.

line 3 A function used for parsing data.

line 4 The function that composes the outgoing message.

line 4 The function that checks the replied message.

line 5 The function that reads the appropriate data file.

line 6 The function that renders the Sudoku puzzle and its answer.

### stim.c

```
1 // standard headers  
2 #include <stdio.h>  
3 #include <stdlib.h>  
4 #include <unistd.h>  
5 #include <time.h>  
6 #include <string.h>  
7 #include <dirent.h>  
8  
9 // stim program headers  
10 #define _ALLOC  
11 #include "stim.h"  
12 #undef _ALLOC  
13 #include "engineMsgs.h"  
14 #include "stimProto.h"  
15  
16 // SIMPL headers  
17 #include "simpl.h"  
18  
19 // for the logger  
20 #define _ALLOC  
21 #include "loggerVars.h"
```

```
22 #undef _ALLOC
23 #include "loggerProto.h"
24
25 /*=====
26   stim - entry point
27 =====*/
28 int main(int argc, char **argv, char **envp)
29 {
30     static char *fn = "stim";
31     int x_it = 0;
32     char line[80];
33     SUDOKU_MSG *in = (SUDOKU_MSG *)inArea;
34     SUDOKU_MSG *out = (SUDOKU_MSG *)outArea;
35     int msgSize;
36
37 // initialize variables, ready program for SIMPL communications etc.
38 initialize(argc, argv);
39
40 while (!x_it) // loop indefinitely
41 {
42     inset = watchset;
43     select(maxfd, &inset, NULL, NULL, NULL);
44
45 // is this from the keyboard?
46 if (FD_ISSET(my_fds[0], &inset))
47 {
48     fgets(line, 79, stdin);
49     line[strlen(line) - 1] = 0;
50
51     switch (line[0])
52     {
53         // help
54         case '?':
55             printf("stimulator commands:\n");
56             printf("f <puzzlefile> - send a test puzzle\n");
57             printf("l - list test puzzles\n");
58             printf("q - quit\n");
59             break;
60
61         // compose and send test puzzle
62         case 'f':
63         {
64             char *p;
65             p = skipOverWhiteSpace(line);
66
67             // build message going to sim/engine
```

```
68     makeMsg(p, out);
69
70     // send the message
71     msgSize = sizeof(int) + size * size * sizeof(char);
72     if (Send(recvID, out, in, msgSize, msgSize) == -1)
73     {
74         fcLogx(FILE_, fn, STIM_MARK, STIM_MARK,
75             "%s: cannot send-%s\n", fn, whatsMyError());
76         printf("%s: cannot send-%s\n", fn, whatsMyError());
77         exit(-1);
78     }
79
80     // check reply from sim/engine
81     checkMsg(in);
82     }
83     break;
84
85     // list test puzzles
86 case 'l':
87     {
88     DIR *dir;
89     struct dirent *mydirent;
90
91     dir = opendir(datapath);
92     if (dir != NULL)
93     {
94         while ((mydirent = readdir(dir)) != NULL)
95         {
96             if ((mydirent->d_name[0] != '.') && (mydirent->d_name[0] != 'C'))
97             {
98                 printf("%s\n", mydirent->d_name);
99             }
100         }
101     }
102     }
103     break;
104
105
106     // quit
107 case 'q':
108     x_it = 1;
109     break;
110
111 default:
112     printf("%s: unknown keypress <%c>\n", fn, line[0]);
113     break;
```

```

114     }
115
116     printf("-> ");
117     fflush (stdout );
118 }
119 } // end of while loop
120
121 // program is finished
122 fcLogx( _FILE_ , fn , globalMask , STIM_MISC , "done" );
123
124 // remove SIMPL attachments
125 name_detach ();
126
127 // exit gracefully
128 exit (0);
129 }
```

lines 2-7 Required standard ‘C’ headers.

lines 10-14 Headers described above. Note the use of \_ALLOC in determining the extern status of global variables used in multiple source code files.

line 17 Required SIMPL definitions.

lines 20-23 Required headers for use with trace logger function calls. .

lines 30-35 Local to main() variable declarations.

line 38 All required initialization procedures are performed here including the all-important SIMPL ***name\_attach***. As well, stim is a sender and so the ***name\_locate*** call for the receiver program is also present. The initialize() function source is contained in the stimInit.c file.

lines 40-119 An infinite loop indicating that the canned messages are to be selected by the user and sent to the receiver, processed and the reply dealt with indefinitely.

lines 42-49 This code deals with receiving instructions from the user via the keyboard.

lines 51-114 The keyboard entries being:

1. ? -help
2. f -the canned message to be sent

3. l -a list of the canned file messages that may be sent
4. q -quit the program

line 68 The makeMsg() function builds the desired outgoing message to the receiver.

line 71 The size of the message is determined. This is necessary for the **Send** function call. Since the reply message is of the same structure, the same value for size will be passed in as the expected reply message size.

lines 72-78 The canned message is sent to the receiver using the SIMPL identification number, the message to the receiver is pointed to by *out*, the anticipated reply message is pointed to by *in* and both the sent and replied message sizes are given by *msgSize*.

line 81 The replied message from the receiver is handled by the function checkMsg().

Let's finish up by looking at the balance of the pertinent code contained in stim.c.

```
1 void makeMsg(char *filename , SUDOKUMSG *out)
2 {
3 // size is global
4 // type is global
5 // array is global
6 char *me = "stim";
7 char file[80];
8
9 // make the file path and name
10 sprintf(file , "%s/%s" , datapath , filename );
11
12 // use data file name to decide on the course of action taken
13 if (!strcmp(filename , "valid_9.dat"))
14 {
15 // set the token value
16 out->token = DO_9;
17 // for the message size
18 size = SIZE_9;
19 // type of reply message expected
20 type = DO_9;
21
22 // read the required data into the global array
23 readData(file );
24
25 // for the message
```

```
26     memcpy(out->elements, array, size * size);
27 }
28 else if (!strcmp(filename, "invalid_9.dat"))
29 {
30     // set the token value
31     out->token = DO_9;
32     // for the message size
33     size = SIZE_9;
34     // type of reply message expected
35     type = PUZZLEFAILURE;
36
37     // for the message
38     memset(out->elements, 0, size * size);
39 }
40 else if (!strcmp(filename, "broken_9.dat"))
41 {
42     // set an invalid token
43     out->token = BADTOKEN;
44     // for the message size
45     size = SIZE_9;
46     // type of reply message expected
47     type = ERROR;
48 /*
49 we do not bother setting the values of the element array
50 and so we will be sending garbage, but it doesn't matter
51 because sim or engine should catch the bad token value first
52 */
53 }
54 else if (!strcmp(filename, "valid_16.dat"))
55 {
56     // set the token value
57     out->token = DO_16;
58     // for the message size
59     size = SIZE_16;
60     // type of reply message expected
61     type = DO_16;
62
63     // read the required data into the global array
64     readData(file);
65
66     // for the message
67     memcpy(out->elements, array, size * size);
68 }
69 else if (!strcmp(filename, "invalid_16.dat"))
70 {
71     // set the token value
```

```

72     out->token = DO_16;
73     // for the message size
74     size = SIZE_16;
75     // type of reply message expected
76     type = PUZZLE_FAILURE;
77
78     // for the message
79     memset(out->elements, 0, size * size);
80 }
81 else if (!strcmp(filename, "broken_16.dat"))
82 {
83     // set an invalid token
84     out->token = BAD_TOKEN;
85     // for the message size
86     size = SIZE_16;
87     // type of reply message expected
88     type = ERROR;
89 /*
90 we do not bother setting the values of the element array
91 and so we will be sending garbage, but it doesn't matter
92 because sim or engine should catch the bad token value first
93 */
94 }
95 else
96 {
97     printf("%s: file name %s is non-sequitor\n", me, filename);
98     exit(-1);
99 }
100 }
```

line 10 The name and path of the canned message file is set.

lines 32-94 Based on the input data file name, we set the size of the puzzle, the message token, the expected reply type (DO\_9 or DO\_16 for success, PUZZLE\_FAILURE for failure and ERROR for a bad token) and the puzzle element array values as necessary.

```

1 void checkMsg(SUDOKU_MSG *in)
2 {
3     // size is global
4     // type is global
5     char answerFile[200];
6
7     // the expected incoming token and the type should match
8     if (in->token != type)
```

```
9  {
10   printf("Problem: type=%d != token=%d\n", type, in->token);
11   return;
12 }
13
14 switch (in->token)
15 {
16 case DO_9:
17   // success
18   // print outgoing array of numbers
19   printf("\nSent puzzle\n");
20   printArray();
21   sprintf(answerFile, "%s/%s", datapath, "answer_9.dat");
22   // read in the canned answer
23   readData(answerFile);
24   // compare the canned answer to the replied solution
25   if (!memcmp(array, in->elements, size * size))
26   {
27     printf("Successful answer-communications intact.\n");
28     printArray();
29   }
30 else
31 {
32   printf("Unsuccessful answer-communication problems?\n");
33 }
34 break;
35
36 case DO_16:
37   // success
38   // print outgoing array of numbers
39   printf("\nSent puzzle\n");
40   printArray();
41   sprintf(answerFile, "%s/%s", datapath, "answer_16.dat");
42   // read in the canned answer
43   readData(answerFile);
44   // compare the canned answer to the replied solution
45   if (!memcmp(array, in->elements, size * size))
46   {
47     printf("Successful answer-communications appear intact.\n");
48     printArray();
49   }
50 else
51 {
52   printf("Unsuccessful answer-communication problems?\n");
53 }
54 break;
```

```
55
56 case ERROR:
57     // sent token is bad or other problem
58     printf("Token failure: which is the correct response.\n");
59     break;
60
61 case PUZZLE_FAILURE:
62     // an invalid puzzle
63     printf("Puzzle failure: which is the correct response.\n");
64     break;
65
66 default:
67     printf("Invalid reply token.\n");
68     break;
69 }
70 }
```

lines 8-12 The reply token is checked to see if it is the expected value. If not, there is a real problem and not a planned one.

lines 14-68 The veracity of the replied message is checked to ensure that the programs are functioning properly and that the message passing is working correctly.

## 14.7 Testing Sim and Stim

In this section we would like to run the simulator and stimulator programs together. Once things are compiled and linked we will have the sim and stim binaries respectively. We want to unit test the two programs with each other so that we can verify their abilities to handle normal and abnormal situations.

Unfortunately, unit testing sim and stim could be a confliction in terms. We must test both with one and other. When we are satisfied that they both work correctly, then we can use them to unit test the GUI programs and the engine. Our testing for sim and stim can be pictured in Figure 12.1. We have previously defined this test to be **sb000**.

Test **sb000** is concerned with checking various aspects of the message passing between stim and sim. The scenarios we are going to run are as follows:

1. Sending a valid 9x9 canned message and replying a valid canned response.
2. Sending an invalid 9x9 message via corrupt puzzle element data, simulating message corruption, and replying a failure.

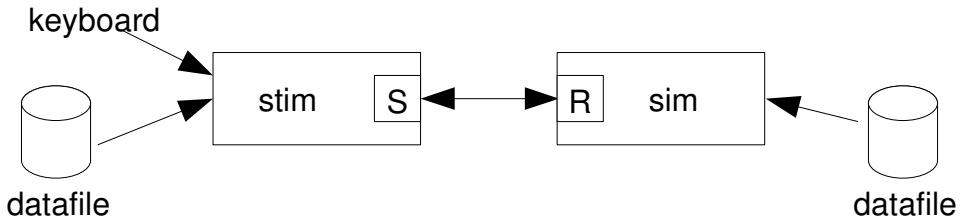


Figure 14.1: Sim/Stim SIMPL Test sb000

3. Sending an invalid 9x9 message via an unknown token.
4. Sending a valid 16x16 canned message and replying a valid response.
5. Sending an invalid 16x16 message via corrupt puzzle element data and replying a failure.
6. Sending an invalid 16x16 message via an unknown token.

In *scenario 1* we are going to send the canned 9x9 puzzle clues to the sim and expect the appropriate canned solution by replied message. We expect success and if success is the response, well and good. If not, there may be a problem with the contents of the message structure or how it is loaded or how it is read. Perhaps there is even a problem with the logic in one or both sim and stim. There may even be a problem with the way the token is set by stim and interpreted by sim. Another probable area for problems is the memory logic pertaining to the messages.

In *scenario 2* we are sending a message with bad 9x9 data. We expect a reply message containing a failure.

In *scenario 3* we are deliberately sending a 9x9 message with an unknown message token and verifying that the sim program handles it correctly.

In *scenario 4* we are simply repeating case 1 but with a 16x16 puzzle and so we are then exercising that particular logic stream.

In *scenario 5* we are sending a message with bad 16x16 data. We expect a reply message containing a failure.

In *scenario 6* we are deliberately sending a 16x16 message with an unknown message token and verifying that the sim program handles it correctly.

Recall from Chapter 9 describing the SIMPL testing framework that we view all tests as being composed of three stages, each represented by a wrapper script.

1. pretest - The setup of the testing environment.
2. dotest - The actual execution of the test.
3. posttest - The examination of the results.

In this test (sb000) we don't require any special setup, hence the first wrapper script called pretest does nothing. The remaining two command scripts will fulfill the sb000 test procedure.

***dotest:*** The dotest script performs the actual tests by running sim and stim via the *runttest* file. The runtest file appears as follows:

```
#!/bin/bash

TEST_NO=sb000
TEST_DESC="sim/stim unit test"

TEST_DIR=$TEST_HOME/testing/test$TEST_NO
OUTFILE=$TEST_DIR/results/test.out

echo "Starting up test #$TEST_NO" | tee $OUTFILE
echo $TEST_DESC | tee -a $OUTFILE
date | tee -a $OUTFILE

cd $SIMPL_HOME/bin
fclogger -n LOGGER > $TEST_DIR/results/junk &

cd $TEST_HOME/sudoku/bin
sim -n SIM -p $TEST_DIR/data -l LOGGER &
sleep 1
stim -n STIM -r SIM -p $TEST_DIR/data -l LOGGER | tee -a $OUTFILE

fcslay SIM
fcslay LOGGER

echo "Test finished you can run posttest $TEST_NO for result"
```

As you can see by inspecting the above coded instructions, sim is run in the background, stim is run in foreground, and the output of the programs is written to an output file called test.out.

***posttest:*** Finally, by running posttest the contents of the dotest output file *test.out* can be examined. When we are certain that both the sim and the stim handle all their messages correctly, we will then be in a position to use them as

a starting point for developing the engine and the GUI programs respectively and to act as test stubs for the engine and GUI program unit tests.

## 14.8 Summary

In this chapter we have constructed the simulator (sim) and stimulator (stim) stub programs derived from the softwareICs repository code. We have modified the repository code to fit the requirements of the Sudoku software. We have plumbed out our *unit* test (sb000) for the sim and stim to verify that they function correctly. As a developer's personal SIMPL toolbox grows, reliance on the softwareICs repository will decrease.

## 14.9 Next Chapter

In the next chapter we are going to take the functional sim code and port it easily to the fully functional Sudoku engine program. We will then use the stim to unit test the engine.

# Chapter 15

## Sudoku Engine Program

In this chapter we are going to describe the process for writing the Sudoku engine program code. We will leverage off our previous work on the sim. By adopting this process, we can be relatively certain that the SIMPL aspect of the engine code will be functional at the start. The engine can be fully tested with our stim. The GUI programming can proceed independently because it can be unit tested with the sim.

### 15.1 Building the Engine Program

What is missing from the sim code that we require for the engine? The answer is the parts that have nothing to do with SIMPL. By adding the problem solving algorithm to the sim we should have the complete engine. We are not going to discuss the algorithm used by the engine to solve Sudoku puzzles in any detail. Suffice to say that the code first tries to solve the puzzle deterministically if there is enough information. If not, then guesses are made recursively after the last point of certitude. The code for this algorithm is freely available on the SIMPL book website.<sup>1</sup>

Let's take a look at the engine code here. It helps to look back at the sim code and note how the sim code has formed the framework for the engine program. We will start with *engine.h* and *engineProto.h* which are the header files containing relevant definitions, global variables, and function prototypes.

---

<sup>1</sup>If you wish to follow along with some real code in front of you the entire Sudoku project code is available at <http://www.icanprogram.com/simplBook>.

engine.h

```

1 // defined macros
2 #define SIZE_9          9
3 #define SIZE_16         16
4 #define SUB_SIZE_9      3
5 #define SUB_SIZE_16     4
6 #define SPACE           ,
7 #define FIRST           1
8 #define ALL             0
9 #define MAX_GUESSES    3
10 #define MAX_MSG_SIZE   1024
11
12 // for the logger
13 #define RECV_MARK      0x00000001
14 #define RECV_FUNC_IO    0x00000002
15 #define RECV_MISC       0x00000010
16
17 typedef struct
18 {
19     int number;
20     int place[SIZE_16]; // maximum possible size
21 } POSSIBILITY;
22
23 // global variables
24 ALLOC int size; // size of the incoming puzzle (9x9 or 16x16)
25 ALLOC char array[SIZE_16 * SIZE_16]; // maximum size
26 ALLOC int subSize; // incoming puzzle's submatrix size (3x3 or 4x4)
27 ALLOC char numbers_9[SIZE_9]; // possible values for 9x9
28 ALLOC char numbers_16[SIZE_16]; // possible values for 16x16
29 ALLOC char *numbers; // pointer to a number array
30 ALLOC unsigned int globalMask; // for the logger
31
32 // for the incoming and outgoing messages
33 ALLOC char inArea[MAX_MSG_SIZE];
34 ALLOC char outArea[MAX_MSG_SIZE];

```

lines 2-10 Definitions used by engine code.

lines 13-15 Definitions used in trace logger function calls.

lines 24-30 Global variable declarations.

lines 33-34 Global incoming and outgoing message buffer declarations.

engineProto.h

```
1 void initialize(int , char **);
2 void myUsage(void);
3 int doCalculations(void);
4 int process(void);
5 int checkArray(void);
6 int doExact(void);
7 int doRows(void);
8 int doColumns(void);
9 int doSubs(void);
10 int findRowPosition(int , char );
11 int findColumnPosition(int , char );
12 int findSubPosition(int , int , char , int *, int *);
13 int doInexact(int );
14 void getPossibilities(int , char , POSSIBILITY *);
15 int findCharacter(int , int , char , int );
16 int checkValues(void);
17 int runTest(int );
18 void printArray(void);
19 void initialize_1(void);
20 void initialize_2(void);
21 void initialize_3(void);
22 void initialize_4(void);
23 void initialize_5(void);
24 void initialize_6(void);
25 void initialize_7(void);
26 void initialize_8(void);
```

lines 1-26 Function prototype declarations.

engine.c

```
1 // standard headers
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <time.h>
6 #include <string.h>
7
8 // engine program headers
9 #define _ALLOC
10 #include "engine.h"
11 #undef _ALLOC
12 #include "engineMsgs.h"
```

```
13 #include "engineProto.h"
14
15 // SIMPL headers
16 #include "simpl.h"
17
18 // for the logger
19 #define _ALLOC
20 #include "loggerVars.h"
21 #undef _ALLOC
22 #include "loggerProto.h"
23
24 /*=====
25   engine - entry point
26 =====*/
27 int main(int argc, char **argv, char **envp)
28 {
29     static char *fn = "engine";
30     int x_it = 0;
31     int nbytes;
32     char *sender;
33     SUDOKU_MSG *in = (SUDOKU_MSG *)inArea;
34     SUDOKU_MSG *out = (SUDOKU_MSG *)outArea;
35     int msgSize;
36     int ret;
37
38 // initialize variables, SIMPL etc.
39 initialize(argc, argv);
40
41 // log the startup
42 fcLogx(FILE, fn, globalMask, RECVMARK, "starting");
43
44 while (!x_it)
45 {
46     // receive incoming messages
47     nbytes = Receive(&sender, inArea, MAX_MSG_SIZE);
48     if (nbytes == -1)
49     {
50         fcLogx(FILE, fn, RECVMARK, RECVMARK,
51             "Receive error-%s", whatsMyError());
52         continue;
53     }
54
55     // decide course of action based on value of the message token
56     switch (in->token)
57     {
58         // a 9x9 puzzle
```

```
59     case DO_9:
60         // set necessary variables
61         size = SIZE_9;
62         subSize = SUB_SIZE_9;
63         memcpy( array , in->elements , SIZE_9 * SIZE_9 );
64         numbers = numbers_9;
65
66         // try to solve the puzzle and compose reply message
67         ret = doCalculations();
68         if (ret != -1)
69         {
70             // success
71             out->token = DO_9;
72             memcpy( out->elements , array , size * size );
73             // use msgSize NOT sizeof(MSG), C may pad out structure
74             msgSize = sizeof(int) + size * size * sizeof(char);
75         }
76     else
77     {
78         // failure
79         out->token = PUZZLE_FAILURE;
80         msgSize = sizeof(int);
81     }
82
83     // reply results to sender
84     if (Reply(sender , out , msgSize) == -1)
85     {
86         fcLogx( __FILE__ , fn , RECV_MARK , RECV_MARK ,
87             "Reply error-%s" , whatsMyError() );
88     }
89     break;
90
91     // a 16x16 puzzle
92     case DO_16:
93         // set necessary variables
94         size = SIZE_16;
95         subSize = SUB_SIZE_16;
96         memcpy( array , in->elements , SIZE_16 * SIZE_16 );
97         numbers = numbers_16;
98
99         // try to solve the puzzle and compose reply message
100        ret = doCalculations();
101        if (ret != -1)
102        {
103            // success
104            out->token = DO_16;
```

```

105     memcpy( out->elements , array , size * size );
106     // use msgSize NOT sizeof(MSG) , C may pad out structure
107     msgSize = sizeof(int) + size * size * sizeof(char);
108     }
109   else
110     {
111     // failure
112     out->token = PUZZLE_FAILURE;
113     msgSize = sizeof(int);
114     }
115
116     // reply results to sender
117   if (Reply(sender , out , msgSize) == -1)
118     {
119     fcLogx( __FILE__ , fn , RECV_MARK , RECV_MARK ,
120             "Reply error-%s" , whatsMyError() );
121     }
122   break;
123
124     // unknown message token
125   default:
126     // reply error condition to sender
127     out->token = ERROR;
128     msgSize = sizeof(int);
129     if (Reply(sender , out , msgSize) == -1)
130       {
131       fcLogx( __FILE__ , fn , RECV_MARK , RECV_MARK ,
132               "Reply error-%s" , whatsMyError() );
133       }
134     fcLogx( __FILE__ , fn , RECV_MARK , RECV_MARK ,
135             "unknown token=%d" , in->token );
136     break;
137   }
138 }
139
140 // program is finished
141 fcLogx( __FILE__ , fn , globalMask , RECV_MARK , "done" );
142
143 // remove SIMPL attachments
144 name_detach();
145
146 // exit gracefully
147 exit(0);
148 }
```

lines 2-6 Required standard ‘C’ headers.

lines 9-13 Headers described above. Note the use of `_ALLOC` in determining the extern status of global variables used in multiple source code files.

line 16 Required SIMPL definitions.

lines 19-22 Required headers for use with trace logger function calls.

lines 29-36 Local to `main()` variable declarations.

line 39 All required initialization procedures are performed here including the all-important SIMPL `name_attach`. The `initialize()` function source is contained in the `engineInit.c` file.

line 42 Inform the trace logger that the engine is starting.

lines 44-138 An infinite loop indicating that messages are to be received, processed and replied to indefinitely.

lines 47-53 The ***Receive*** blocks until a message arrives. The sender's SIMPL identity is pointed to by `sender`, the message is placed into the memory pointed to by `inArea` and the message cannot be any larger than the memory area indicated by `MAX_MSG_SIZE`. Note the error checking.

lines 56-138 We look at the value of the incoming message token in order to decide on the correct course of action. Note the switch default which catches errors in the token value.

lines 67,100 A solution to the incoming puzzle is attempted via the `doCalculation()` function call.

lines 84,117,129 The reply message is sent to the sender program via ***Reply***. The message contents are set based on the findings of `doCalculation()` function or due to a bad message token.

Comparing this code to the sim code shows a line for line correlation at many points. Many SIMPL projects are evolved in exactly this manner; clone some existing code and modify it to suit. The real functional differences between the engine and the sim code are largely isolated in the transition from the ***process*** call in `sim.c` to the ***doCalculation*** call in `engine.c`.

Here is a quick look at the rest of the more relevant engine code.

*pertinent engine.c code*

```

1 int doCalculations(void)
2 {
3 // check incoming puzzle values
4 if (checkValues() == -1)
5 {
6     return(-1);
7 }
8
9 // let's try determinism
10 if (doExact() == 0)
11 {
12     return(0);
13 }
14
15 // let's make a guess
16 if (doInexact(MAX_GUESSES) == 0)
17 {
18     return(0);
19 }
20
21 return(-1);
22 }
```

lines 4-7 The checkValues() function checks the incoming element data for inconsistencies.

lines 10-13 The doExact() function is the entry point for attempting a strictly deterministic solution to the puzzle.

lines 16-19 The doInexact() function tries to arrive at a solution through a series of recursive guesses if the deterministic method fails.

The code for doExact() and doInexact() are not printed in this book as they have nothing directly to do with SIMPL. The code can be found in the engineUtils.c file available in the source tarball.

## 15.2 Testing the Engine

Supposing at this point that we have an engine program that compiles and links, we are now ready for unit testing with the stim. Unit testing the engine can be

pictured in Figure 15.1.

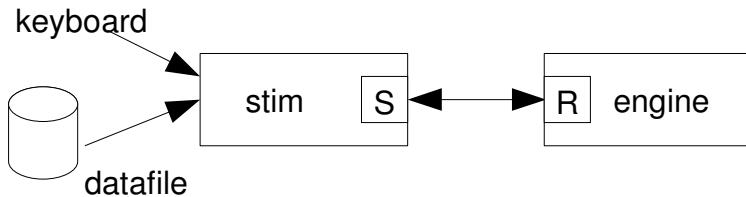


Figure 15.1: Engine Unit Test sb001

Test **sb001** is concerned with checking various aspects of the message passing between stimulator and engine. These tests are identical to the sim/stim tests in the previous chapter. Stim will test the SIMPL functionality of the engine. Moreover, unlike sim which had canned responses, the engine will have to solve the valid puzzles sent to it. For completeness we repeat some of the verbiage in the sim/stim chapter with respect to the various scenarios we will employ.

1. Sending a valid 9x9 canned message and replying a solution.
2. Sending an invalid 9x9 message via corrupt puzzle element data, simulating message corruption, and replying a failure.
3. Sending an invalid 9x9 message via an unknown token.
4. Sending a valid 16x16 canned message and replying a solution.
5. Sending an invalid 16x16 message via corrupt puzzle element data and replying a failure.
6. Sending an invalid 16x16 message via an unknown token.

In *scenario 1* we are going to send the canned 9x9 puzzle clues to the engine and expect a puzzle answer which should correspond to the canned solution by replied message. We expect success and if success is the response then the Sudoku engine is able to solve puzzles. If not, there may be a problem with the contents of the message structure or how it is loaded or how it is read or our puzzle engine is actually dysfunctional. Another probable area for problems is the memory logic pertaining to the messages.

In *scenario 2* we are sending a message with bad 9x9 data. We expect a reply message containing a failure.

In *scenario 3* we are deliberately sending a 9x9 message with an unknown message token and verifying that the engine program handles it correctly. This should work immediately because it is the same code as in the sim program.

In *scenario 4* we are simply repeating test 1 but with a 16x16 puzzle and so we are then exercising that particular logic stream.

In *scenario 5* we are sending a message with bad 16x16 data. We expect a reply message containing a failure.

In *scenario 6* we are deliberately sending a 16x16 message with an unknown message token and verifying that the engine program handles it correctly.

As in the sim/stim chapter, the following command scripts comprise test sb001:

- dotest sb001
- posttest sb001

*dotest*: The dotest script performs the actual tests by running engine and stim via the runtest file. The runtest file appears as follows:

```
#!/bin/bash

TEST_NO=sb001
TEST_DESC="engine unit test"

TEST_DIR=$TEST_HOME/testing/test$TEST_NO
OUTFILE=$TEST_DIR/results/test.out

echo "Starting up test #$TEST_NO" | tee $OUTFILE
echo $TEST_DESC | tee -a $OUTFILE
date | tee -a $OUTFILE

cd $SIMPL_HOME/bin
fclogger -n LOGGER > $TEST_DIR/results/junk &

cd $TEST_HOME/sudoku/bin
engine -n ENGINE -l LOGGER &
sleep 1
stim -n STIM -r ENGINE -p $TEST_DIR/data -l LOGGER | tee -a $OUTFILE

fcslay ENGINE
fcslay LOGGER

echo "Test finished you can run posttest $TEST_NO for result"
```

Again, as in the stim/sim test you can see by inspecting the above coded instructions that engine is run in the background, stim is run in foreground, and the output of the programs is written to an output file called *test.out*.

*posttest*: Running posttest, the contents of the dotest output file *test.out* may be examined. When we are satisfied that engine works adequately, we are now in a position to move on to looking at the GUI programming.<sup>2</sup>

## 15.3 Self Test

We stated earlier that the actual processing code for the Sudoku engine is not included in the book. However, for those interested in testing this code there is a self-test capability inherent in the engine software. The self-test code is called from the *engineInit.c* file through function call *runTest()*. The self-test is activated when the engine is run in standalone mode and is started from the command line as follows:

```
engine -t num
```

where num=1-8. These numbers represent various self-tests, some of which are quite difficult and may require several minutes to solve.

## 15.4 Summary

In this chapter we built the Sudoku engine program based directly on the sim program software. We also based its unit test *sb001* on the unit test used for the sim and stim, *sb000*. If all is well we have a fully functioning Sudoku engine that is capable of solving 9x9 and 16x16 puzzles. We are now in a position to move on to the GUI software.

## 15.5 Next Chapter

In the next chapter we are going to develop the Python GUI secure in the knowledge that we have functional sim and engine receivers with which to measure our new GUI software against.

---

<sup>2</sup>Samples of these testing framework scripts can be found in the project tarball for the Sudoku project code at <http://www.icanprogram.com/simplBook>.



# Chapter 16

## Python/Tk GUI

### 16.1 The Python GUI

Our hypothetical engine development team has been able to evolve the simulator code through a logical progression of stages into our Sudoku puzzle engine. They were able to do this in part because they had a capable GUI stub in the form of a stimulator to exercise the engine code changes as they were developed. The Python GUI team will not get off as easily. While the stimulator code is functionally the same as what the GUI will require, it is written in a very different style (procedural vs. OOP) and in a very different programming language ('C' vs. Python). Furthermore, the SIMPL messaging aspects of our Python GUI will likely only form a minor portion of the actual code. As such the GUI team will not be able to use the stimulator as seed code and evolve it in a logical progression into their GUI solution. What the Python GUI team does have going for it however, is the fact that they need not concern themselves with any aspect of the engine algorithm other than conforming to the SIMPL messaging interface embodied by the sim. These SIMPL aspects of the GUI code will be straightforward and the GUI team will be able to code and debug these early on and spend the bulk of their effort on the GUI itself.

As with most GUI code, the bulk of the lines of code will be concerned with interface details such as window/widget size and position. More details are involved with displaying a GUI representation of the Sudoku puzzle than were needed for the straight stimulator test stub. Font selection, point size, colour schemes and widget layout are just some of the things that the GUI design team needs to be working out. In addition, the GUI design team needs to be thinking about the human interface to the GUI as well. Things like how will the

puzzle elements get entered and how the puzzle engine gets engaged need to be designed. As the team works through these items a proposed screen layout is the likely outcome.

There are many possible GUI interface *choices* for our puzzle solver. We have deliberately chosen a very basic one. This is a book about SIMPL and not a book about Python GUI programming after all. The important thing to remember as you examine the details of our basic solution below is that all GUI solutions will follow the basic SIMPL sender layout: ***name\_attach***, ***name\_locate***, ***Send*** and process response. It is also important to remember that SIMPL and our SIMPL Testing Framework has allowed the GUI team to completely isolate their work from that of the engine team. The only point of interface between those teams is in the tokenized SIMPL message definitions, which for the GUI team are fully exercised by the sim. In more realistic real world projects this ability to separate project teams is an important part of the SIMPL way.

## 16.2 Building the Python/Tk GUI

In this section we are going to examine the Python/Tk GUI code. Note that much of the callback code has been left out in order to save space. This code is available as part of the Sudoku source code tarball.<sup>1</sup>

### gui.py

```
1 #! /usr/Python-2.5.1/python
2
3 # import necessary modules
4 import sys
5 import tkMessageBox
6 from Tkinter import *
7 from csimpl import *
8
9 # define constants
10 DO_9=0
11 DO_16=1
12
13 # global variables
14 global matrixElements # a series of button widgets
15 global size           # a 9X9 or 16X16 puzzle matrix
16 global numbers        # the possible puzzle values
17 global saved          # saved values for puzzles
```

---

<sup>1</sup>The Python/Tk code is available at <http://www.icanprogram.com/simplBook>

```
18 """
19 Many of the callbacks have been left out of the book.
20 They would normally appear in this area.
21 """
22
23 # a callback for the submit button
24 # take the button text and use to build the SIMPL message
25 def hndlSubmit(receiverId):
26     # why send an empty array?
27     if hndlCheck():
28         tkMessageBox.showwarning("Warning", "No non-blank elements")
29         return
30
31     # assemble the message
32     if size == 9:
33         outToken = DO_9
34     else:
35         outToken = DO_16
36
37     length = size * size
38
39     # pack the outgoing message token
40     packInt(outToken, SIMPL_BIN)
41
42     for cell in range(length):
43         # pack the outgoing message puzzle elements
44         packChar(matrixElements[cell]["text"], SIMPL_BIN)
45         # save the puzzle elements
46         saved[cell] = matrixElements[cell]["text"]
47
48     # send the message
49     retVal = send(receiverId)
50     if retVal == -1:
51         tkMessageBox.showerror("SIMPL send error", whatsMyError())
52         sys.exit(-1)
53
54     # unpack the reply message token
55     inToken = unpackInt(SIMPL_BIN)
56
57     # not a successful solution?
58     if inToken != outToken:
59         tkMessageBox.showwarning("Warning", "Problem solving puzzle")
60     else:
61         # show results- sent values BLACK, calculated values RED
62         for cell in range(length):
63             matrixElements[cell]["text"] = unpackChar(SIMPL_BIN)
```

```
64     if saved[cell] == " ":
65         matrixElements[cell]["foreground"] = "red"
66 #####
67 # operational start of the program
68
69
70 # check command line
71 if len(sys.argv) != 4:
72     tkMessageBox.showerror("Incorrect command line")
73     sys.exit(-1)
74
75 # initialize some variables
76 sName = sys.argv[1] # this program's SIMPL name
77 rName = sys.argv[2] # the sim/engine program's SIMPL name
78 if sys.argv[3] == "16":# the puzzle size
79     size = 16
80 else:
81     size = 9
82 matrixElements = [None] * size * size # this puzzles elements
83 saved = [None] * size * size # saved values for matrix elements
84
85 # what characters are available for populating the puzzle?
86 if size == 16:
87     numbers = \
88         [ " ", "0" , "1" , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9" , "A" , "B" , \
89         "C" , "D" , "E" , "F" ]
90 else:
91     numbers = [ " " , "1" , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9" ]
92
93 # attach the SIMPL name
94 retVal = nameAttach(sName, 1024)
95 if retVal == -1:
96     err = whatsMyError() + ": check for another program instance"
97     tkMessageBox.showerror("SIMPL name attach error", err)
98     sys.exit(-1)
99
100 # name locate the C program SIMPL receiver
101 receiverId = nameLocate(rName)
102 if receiverId == -1:
103     err = whatsMyError() + ": is the receiver program running?"
104     tkMessageBox.showerror("SIMPL name attach error", err)
105     sys.exit(-1)
106
107 # initialize Tk for graphics
108 root = Tk()
109
```

```

110 # the graphical matrix elements are to be buttons
111 for row in range(size):
112     rowframe = Frame(root)
113     rowframe.pack(fill=BOTH)
114     for column in range(size):
115         num = row * size + column
116         if size == 16:
117             myFont = ("Times", 30, "bold")
118         else:
119             myFont = ("Times", 20, "bold")
120         matrixElements[num] = Button(rowframe, borderwidth=1,
121             relief=SOLID, justify=CENTER, bg="White", fg="Black",
122             text=" ", font=myFont, width=2)
123         matrixElements[num].bind("<Button-1>", hndlIncrease)
124         matrixElements[num].bind("<Button-3>", hndlDecrease)
125         matrixElements[num].pack(side=LEFT)

126 # the bottom frame of buttons
127 rowframe = Frame(root)
128 rowframe.pack(fill=BOTH)
129 Button(rowframe, justify=CENTER, text="Help",
130     command=hndlHelp).pack(side=LEFT)
131 Button(rowframe, justify=CENTER, text="Clear",
132     command=hndlClear).pack(side=LEFT, expand=YES)
133 Button(rowframe, justify=CENTER, text="Save",
134     command=hndlSave).pack(side=LEFT, expand=YES)
135 Button(rowframe, justify=CENTER, text="Restore",
136     command=hndlRestore).pack(side=LEFT, expand=YES)
137 Button(rowframe, justify=CENTER, text="Test Puzzle",
138     command=hndlTest).pack(side=LEFT, expand=YES)
139 Button(rowframe, justify=CENTER, text="Submit",
140     command=(lambda: hndlSubmit(receiverId))).pack(side=RIGHT)

141 # handle user input
142 root.mainloop()

```

line 1 Declares the location of the python interpreter to be used.

lines 4-7 Required imported modules. Note particularly the csimpl module.

This is the SIMPL wrapper module that uses packing and unpacking methods for constructing and deconstructing messages respectively.

lines 10-11 Defined constants.

lines 14-17 Global variable declarations.

lines 25-65 The hndlSubmit callback. This callback has been included in the code listing because it is responsible for packing the message to be sent, sending the message, and then unpacking the reply.

lines 27-29 There is no point in sending an empty array for processing.

lines 32-35 The value of the token for outgoing message is set.

line 37 The size of the data portion of the outgoing message is set.

line 40 The value of the token for the outgoing message is packed into the message.

lines 42-44 The values of the data portion of the message are packed. ie. the numerical puzzle clues.

lines 49-52 The outgoing message is sent to the engine program for a solution to the puzzle.

line 55 The replied message token is unpacked.

lines 58-65 The incoming message token is checked and if valid, the puzzle solution values are unpacked and displayed on the GUI.

lines 71-73 The command line is checked.

lines 76-83 Variable initializations.

lines 94-98 SIMPL ***nameAttach***.

lines 101-105 SIMPL ***nameLocate*** of the receiving process.

line 108 Tk instance is initialized.

lines 111-125 The sudoku puzzle matrix elements are drawn.

lines 128-141 The buttons are drawn.

line 144 Event loop is started for handling user input.

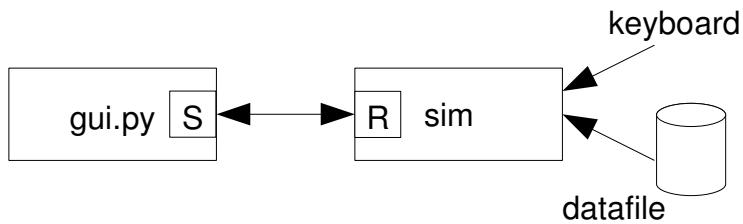


Figure 16.1: Python/Tk Unit Test sb002

## 16.3 Testing the Python/Tk GUI

Now that we have a Python/Tk GUI program, we are now ready to unit test it with the help of the sim program. Unit testing the Python GUI can be pictured in Figure 16.1.

Test *sb002* is concerned with checking the message passing between the python GUI and the sim. Essentially, *sb002* confirms that the GUI client functions correctly. From the earlier test *sb000* we can be relatively certain that the sim works correctly. This allows us an opportunity to work on the GUI program knowing that any problems encountered are most likely due to the Python code. For this test, we simply want to send the canned 9x9 and 16x16 messages to the sim and observe the reply. In order to do this we run the following commands:

- pretest sb002
- dotest sb002
- posttest sb002

*pretest*: We have elected to dedicate the pretest script to choosing the size of the puzzle to be tested, 9x9 or 16x16. Pretest is a wrapper script which points to the *setup* file which looks like:

```

#!/bin/bash

TEST_NO=sb002
TEST_DESC="Python GUI unit test"

TEST_DIR=$TEST_HOME/testing/test$TEST_NO
DATA_DIR=$TEST_DIR/data

OUTFILE=$TEST_DIR/results/test.out

```

```

echo "Starting up test #$TEST_NO" | tee $OUTFILE
echo $TEST_DESC | tee -a $OUTFILE
date | tee -a $OUTFILE

echo ""
echo "====="
echo "1) 9x9 puzzle"
echo "2) 16x16 puzzle"
echo ""
echo -n "-> "
read ans

cd $TEST_DIR/data

if [ $ans == '1' ]
then
    if [ -f size16 ]
    then
        rm size16
    fi
else
    date > size16
fi

echo "Test $TEST_NO setup finished"

```

dotest: The dotest script performs the actual tests involving running the python GUI and the sim. The dotest script is itself a wrapper for the specific STF *runttest* file. In the GUI interface, selecting the *Test Puzzle* button should cause some of the matrix elements to be filled in automatically. These values represent the 9x9 or 16x16 canned test puzzle messages that the sim receives. The *Submit* button activates the submission of this test puzzle to the engine stub (sim). The sim is designed to recognize this test puzzle and return a valid solution message. If any of the values in the test puzzle are altered, this is equivalent to sending an invalid message and the sim should respond with an error.

The runtest file appears as follows:

```

#!/bin/bash

TEST_NO=sb002
TEST_DESC="Python GUI unit test"

TEST_DIR=$TEST_HOME/testing/test$TEST_NO

```

```
OUTFILE=$TEST_DIR/results/test.out

echo "Starting up test #$TEST_NO" | tee $OUTFILE
echo $TEST_DESC | tee -a $OUTFILE
date | tee -a $OUTFILE

cd $SIMPL_HOME/bin
fclogger -n LOGGER > $TEST_DIR/results/junk &

cd $TEST_HOME/sudoku/bin
sim -n SIM -p $TEST_DIR/data -l LOGGER &
sleep 1

if [ -f $TEST_DIR/data/size16 ]
then
    SIZE=16
else
    SIZE=9
fi

python gui.py gui.py SIM $SIZE

fcslay SIM
fcslay LOGGER

echo "Test finished you can run posttest $TEST_NO for result"
```

During the execution of test sb002 the puzzle screen will appear something like Figure 16.2.

The posttest script isn't used in this test because most of the things you'd want to validate are visible only on the GUI screen itself. Internal errors detected by the sim will appear in the trace log.

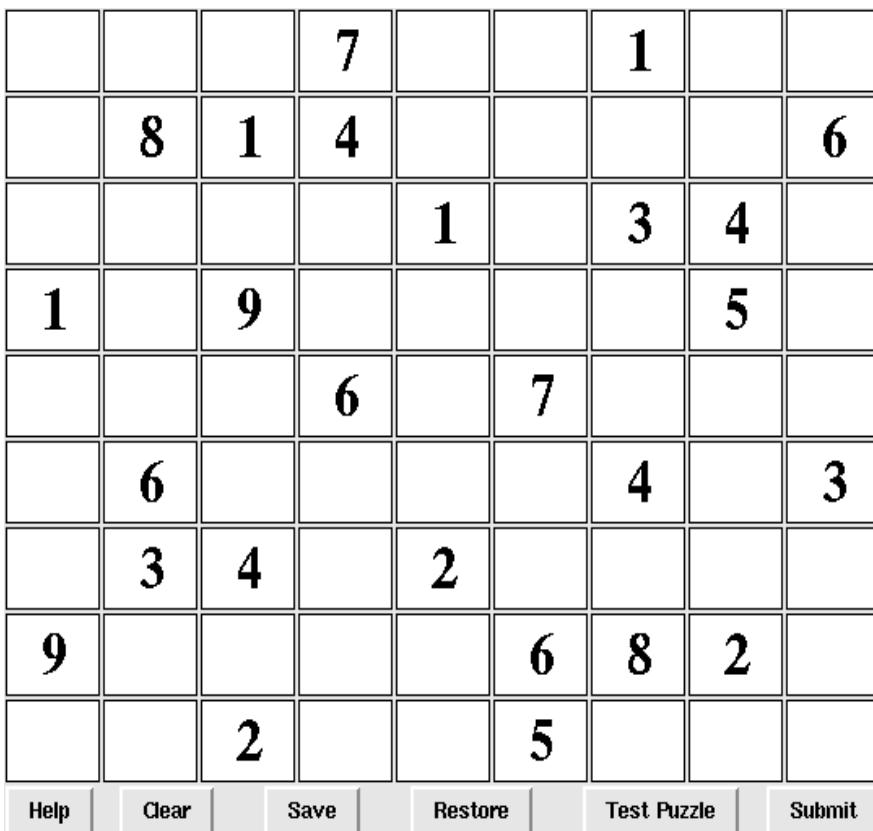


Figure 16.2: Python/Tk GUI screen

## 16.4 Summary

In this chapter we were given the Python GUI and unit tested it with the sim via test sb002. The number of tests was smaller because we have already ascertained that the sim can handle unknown tokens and the GUI tokens are contained within the code.

## 16.5 Next Chapter

In the next chapter we will perform the integration testing of the Python GUI with the engine because we have satisfactorily unit tested both the Python GUI and the engine programs.

# Chapter 17

## Integration Testing - Python GUI/ Sudoku Engine

### 17.1 Python GUI Integration Test

Up to this point in the project we have exercised unit tests associated with the various SIMPL modules in our system. Once those modules are thoroughly unit tested, they need to be joined together as they would be deployed and integration tested in that configuration. Test sb004 represents the integration test with the Python GUI client and the Sudoku engine. See Figure 17.1

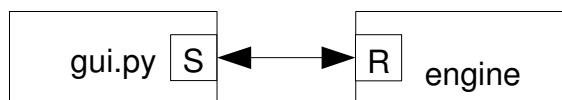


Figure 17.1: Python Integration test

The unit test sb002 validated that the Python client successfully composed a Sudoku puzzle message and submitted it to the engine simulator (sim). So at this stage we know that the client works. Likewise, we know that if the puzzle message is malformed the client correctly handles the error returned. Finally, the engine unit test has demonstrated that the engine can receive puzzles, solve them, and return the solution to the sender.

In the integration test we are interested in checking whether the two halves of the system join together correctly. We are also interested in investigating the

performance of this system in realistic operating conditions. Integration tests are used to expose the system to the end users and to solicit their suggestions for improvements. During integration testing we would want to submit a range of puzzles to the system and observe the ensuing behaviour. Our *simplest system which works* initial design uses a blocking **Send** which will impact the feel of the GUI in cases where the puzzle solving engine is occupied for more than a few seconds. We would want to gauge user reaction in these cases.

This test will formally be run in the normal manner:

- pretest sb004
- dotest sb004
- posttest sb004

The pretest script will allow the tester to decide the size of the puzzle to displayed, 9x9 or 16x16 just as in the unit test (sb002).

The dotest script for sb004 would look something like the version below.

```
#!/bin/bash

TEST_NO=sb004
TEST_DESC="integration test with Python GUI"

TEST_DIR=$TEST_HOME/testing/test$TEST_NO
OUTFILE=$TEST_DIR/results/test.out

echo "Starting up test #$TEST_NO" | tee $OUTFILE
echo $TEST_DESC | tee -a $OUTFILE
date | tee -a $OUTFILE

cd $SIMPL_HOME/bin
fclogger -n LOGGER > $TEST_DIR/results/junk &

cd $TEST_HOME/sudoku/bin
engine -n ENGINE -l LOGGER &
sleep 1

if [ -f $TEST_DIR/data/size16 ]
then
    SIZE=16
else
    SIZE=9
fi
```

```
python gui.py gui.py ENGINE $SIZE  
fcslay ENGINE  
fcslay LOGGER  
echo "Test finished you can run posttest $TEST_NO for result"
```

Note that the engine and gui.py executable files are pulled from the master Sudoku install area (sudoku/bin). This is standard practice for an integration test.<sup>1</sup>

There are several areas we would want to investigate using test sb004.

- We want to send the engine program the test puzzle just as we did in test sb002. The difference is that unlike the sim, the engine program will actually solve the puzzle and reply the solution.
- We would want to enter various Sudoku puzzles from any readily available Sudoku puzzle book. This will allow us to observe that the engine correctly solves the puzzles in a timely manner.
- We will want to deliberately send bad puzzle entries for submission to see if the puzzle engine reports an error.
- We will want to *play*, perhaps by creating some puzzles of our own. One way is to take the test puzzle and remove some of the values or change the values around. The results can be very surprising.

All of the script code associated with test sb004 is available in the accompanying tarball for the book.<sup>2</sup>

In any project of real world complexity, integration tests are by definition complex. This is not the optimal place to debug and solve a code defect. If an error is spotted in the behaviour of the engine or the Python GUI in the integration test phase, the better approach is to first try to replicate the error in the respective unit test (or a derivative of that unit test). It is more cost effective to spend the effort to replicate the error in a well defined unit test environment than in a more complex and less easily controlled integration test. Once the defect is isolated in this manner, the code is repaired and retested in the same

---

<sup>1</sup>The astute reader will recognize that in our simple project the master install area and the local build area are the same. This is not normally the case in more complex projects.

<sup>2</sup>If you wish to follow along with some real code in front of you the entire Sudoku project code is available at <http://www.icanprogram.com/simplBook>.

unit test environment. Only once the defect fix is validated in the unit test is the repaired code rerun in the integration test to close off the sequence.

The STF and the decomposition of an application into SIMPL modules makes this type of testing sequence (unit-integration-defect-unit-integration) easier to manage.

## **17.2 Summary**

In this chapter we discussed the integration test of the Python GUI with the Sudoku engine. If everything ran per spec, we have virtually completed this part of the project because the real world deployment would be very similar to the dotest script with the exception that the GUI and the engine would be run on different host computers.

Whatever other games are to be created, they would follow exactly the same development route that we have mapped out with Sudoku.

The SIMPL paradigm of modular design and the accompanying SIMPL testing framework make debugging complex software easier.

## **17.3 Next Chapter**

In the next chapter we are going to look at what it would take to expand the Sudoku software along various lines such as:

1. forking an engine process per GUI request,
2. running the client GUI and server engine on different hosts within a network,
3. a Tcl/Tk client GUI,
4. a Java/Swing client GUI, and
5. a CGI approach to Sudoku.

# Chapter 18

## Sudoku Evolution

In Chapter 11 we discussed the system picture which would match the project specification. We also discussed the simplest possible subimplementation which had any value and we implemented that first, ie. the Sudoku GUI and puzzle processing engine. The testing framework was built for this simplified system and the code was written and tested. At this point we have working a Sudoku Python client and a working *single threaded* Sudoku puzzle engine.

One of the selling points of the SIMPL approach to application development, is that basic SIMPL implementations can be readily extended to arrive at a more feature rich spec. In this chapter we are going to explore in various levels of detail, extending what we already have to encompass the full project specification.

The evolution of the Sudoku puzzle engine will be in the direction of providing more robust handling of multiple simultaneous client connections. The evolution of the GUI will be to provide multi-language, non-Linux OS and web services options.

### 18.1 Forking Engine

In our initial design the Sudoku engine was a basic SIMPL receiver. Each puzzle is submitted with a blocking **Send** and the puzzle solution is contained in the response. This architecture could support multiple Python GUIs each as straight SIMPL senders. As long as puzzles were kept relatively trivial (ie. engine could solve them quickly) the system would remain usable. Problems could occur with this architecture when a particular user submits a difficult puzzle; in that case the single thread engine could be occupied for unacceptable periods of time solving the difficult Sudoku puzzle. During this interval other users would experience

a delay in getting their puzzle submitted. More importantly, since each GUI is a straight SIMPL sender, this blocking would also freeze each GUI. This is not the desired optimum user experience.

To help alleviate this difficult puzzle bottleneck, it would be desirable to have each puzzle assigned to an independent engine thread. A tried and true approach to such threading is to have each new connection fork a child process and hand off the SIMPL channel to that child (see Figure 18.1).

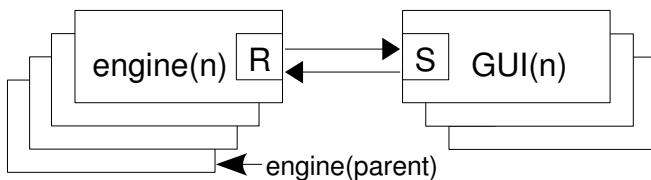


Figure 18.1: Forked Engine

The advantage of a forked engine over the single threaded engine is that each puzzle client will get its own instance of a puzzle engine. Other users (clients) won't get held up if a particular puzzle takes a long time to solve. Within a forked engine architecture, when the puzzle client has completed all of its requests for solutions to Sudoku puzzles, both the client and the forked engine instance will vanish.

A forked engine is still going to be a SIMPL receiver. Much of the original algorithm will be retained intact. The only addition will be the algorithm for forking a child process upon receiving a valid message token. The code for a forking engine follows. Comparison with the original engine code (See Chapter 15) and the following code listing demonstrates the extendability of the SIMPL approach. Note that only the most pertinent differences between the *engine* and *forked engine* code are going to be referred to in the code description.

### forked\_engine.c

```

1 // standard headers
2 #include <errno.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <time.h>
7 #include <string.h>
```

```
8 #include <sys/wait.h>
9
10 // engine program headers
11 #define _ALLOC
12 #include "engine.h"
13 #undef _ALLOC
14 #include "engineMsgs.h"
15 #include "engineProto.h"
16
17 // SIMPL headers
18 #include "simpl.h"
19
20 // for the logger
21 #define _ALLOC
22 #include "loggerVars.h"
23 #undef _ALLOC
24 #include "loggerProto.h"
25
26 // function prototypes
27 void engineChild(char *);
28 void killZombies(void);
29
30 /*—————
31     forked engine — entry point
32—————*/
33 int main(int argc, char **argv, char **envp)
34 {
35     static char *fn = "engine";
36     int x_it = 0;
37     int nbytes;
38     char *sender;
39     pid_t childPid;
40     SUDOKUMSG *out = (SUDOKUMSG *)outArea;
41
42     // initialize variables, SIMPL etc.
43     initialize(argc, argv);
44
45     // log the startup
46     fcLogx(_FILE_, fn, globalMask, RECVMARK, "starting");
47
48     while (!x_it)
49     {
50         // receive incoming messages
51         nbytes = Receive(&sender, inArea, MAX_MSG_SIZE);
52         if (nbytes == -1)
53         {
```

```
54     fcLogx( __FILE__ , fn , RECV_MARK, RECV_MARK,
55             "Receive error-%s" , whatIsMyError () );
56     continue;
57 }
58
59 // fork
60 childPid = fork ();
61 if (childPid < 0) // forking failure
62 {
63     fcLogx( __FILE__ , fn , RECV_MARK, RECV_MARK,
64             "%s: fork error-%s\n" , fn , strerror(errno));
65     // reply error condition to sender
66     out->token = ERROR;
67     Reply(sender , out , sizeof(int));
68 }
69 else if (childPid == 0) // child
70 {
71     engineChild(sender);
72 }
73
74 // parent
75 // check for zombie children
76 killZombies ();
77 }
78
79 return(0);
80 }
81
82 void engineChild(char *sender)
83 {
84     const char *fn = "engineChild";
85     SUDOKU_MSG *in = (SUDOKU_MSG *)inArea;
86     SUDOKU_MSG *out = (SUDOKU_MSG *)outArea;
87     int msgSize;
88     int ret;
89     char me[30];
90
91 // detach from parent's SIMPL
92 child_detach ();
93
94 // create a unique SIMPL name
95 sprintf(me, "%s-%d" , fn , getpid ());
96
97 // SIMPL name attach
98 ret = name_attach(me, NULL);
99 if (ret == -1)
```

```
100  {
101      printf("%s: unable to attach <%s>%s\n", fn, me, whatsMyError());
102      exit(-1);
103  }
104
105 // decide course of action based on the value of the message token
106 switch (in->token)
107 {
108     // a 9X9 puzzle
109     case DO_9:
110         // set necessary variables
111         size = SIZE_9;
112         subSize = SUB_SIZE_9;
113         memcpy(array, in->elements, SIZE_9 * SIZE_9);
114         numbers = numbers_9;
115         // try to solve the puzzle and compose reply message
116         ret = doCalculations();
117         if (ret != -1)
118         {
119             // success
120             out->token = DO_9;
121             memcpy(out->elements, array, size * size);
122             // use msgSize NOT sizeof(MSG) because C may pad out structure
123             msgSize = sizeof(int) + size * size * sizeof(char);
124         }
125     else
126     {
127         // failure
128         out->token = PUZZLE_FAILURE;
129         msgSize = sizeof(int);
130     }
131
132     // reply results to sender
133     if (Reply(sender, out, msgSize) == -1)
134     {
135         fcLogx(FILE_, me, RECV_MARK, RECV_MARK,
136             "Reply error-%s", whatsMyError());
137     }
138     break;
139
140 // a 16X16 puzzle
141 case DO_16:
142     // set necessary variables
143     size = SIZE_16;
144     subSize = SUB_SIZE_16;
145     memcpy(array, in->elements, SIZE_16 * SIZE_16);
```

```
146 numbers = numbers_16;
147
148 // try to solve the puzzle and compose reply message
149 ret = doCalculations();
150 if (ret != -1)
151 {
152     // success
153     out->token = DO_16;
154     memcpy(out->elements, array, size * size);
155     // use msgSize NOT sizeof(MSG) because C may pad out structure
156     msgSize = sizeof(int) + size * size * sizeof(char);
157 }
158 else
159 {
160     // failure
161     out->token = PUZZLE_FAILURE;
162     msgSize = sizeof(int);
163 }
164
165 // reply results to sender
166 if (Reply(sender, out, msgSize) == -1)
167 {
168     fcLogx(FILE_, me, RECV_MARK, RECV_MARK,
169             "Reply error-%s", whatsMyError());
170 }
171 break;
172
173 // unknown message token
174 default:
175     // reply error condition to sender
176     out->token = ERROR;
177     msgSize = sizeof(int);
178     if (Reply(sender, out, msgSize) == -1)
179     {
180         fcLogx(FILE_, me, RECV_MARK, RECV_MARK,
181                 "Reply error-%s", whatsMyError());
182     }
183     fcLogx(FILE_, me, RECV_MARK, RECV_MARK,
184             "unknown token=%d", in->token);
185     break;
186 }
187
188 // program is finished
189 fcLogx(FILE_, me, globalMask, RECV_MARK, "done");
190
191 // remove SIMPL attachments
```

```
192 name_detach();  
193 // exit gracefully  
194 exit(0);  
195 }  
196  
197 void killZombies()  
{  
register int i;  
pid_t pid;  
202  
for (i = 0; i < 10; i++)  
{  
    pid = waitpid(-1, NULL, WNOHANG | WUNTRACED);  
    if (pid <= 0)  
    {  
        break;  
    }  
}  
211 }
```

lines 27-28 New function prototypes related to the forking.

lines 51-57 Puzzle messages are received from the GUI programs.

lines 60-72 After a puzzle message is received, the program forks into a parent and a child. The child process will have all of the parent's data including the newly received puzzle message via a copy of the global memory that was created during the forking procedure.

line 71 The child process continues from this point by calling the engineChild() function.

line 76 The parent process continues by *waiting out* any dead but not forgotten children. Following this crucial step, the parent loops around to await another incoming puzzle message.

line 82 The actual problem solving aspects of the original engine program are contained within this function which forms the body of the child process.

line 92 The child\_detach() call is very important. When the original program forked into parent/child processes, the parent's SIMPL attributes were copied to the child. These attributes must *not* be shared. The child\_detach() releases the parent's SIMPL attributes from the child.

lines 98-103 The child performs a SIMPL name attach in order to have its own SIMPL communication channels. Note that the SIMPL name used is meant to be unique, hence a name based upon its process id which must also be unique. See line 95 above.

lines 106-196 As in the original program, an attempt is made to solve the puzzle and have the results replied back to the GUI program. The child process knows which sender to reply to as the sender's SIMPL id (set upon the parent's `Receive()`) was passed to the child upon its creation. Note that the child exits after having performed its function.

lines 198-211 The `killZombies()` function is called by the parent process in order to ensure that any dead but not forgotten (zombie) child processes do not linger and take up system resources.<sup>1</sup>

While the forking SIMPL receiver will help with load balancing, it doesn't totally skirt the difficult puzzle issue. Since the GUI is still a blocking sender, the user with the difficult puzzle will still experience GUI freeze while the puzzle is solved. To attack that problem we need to revert our GUI to a straight SIMPL receiver and flip the message direction by converting the forking SIMPL engine (receiver) into a forking SIMPL agent (sender). In Section 6.8 we saw that there is little difference between processing a SIMPL message which is captured via the ***Receive*** function and a SIMPL message captured by the ***Reply*** function. So flipping the engine around from a receiver to an agent involves little more than changing a couple of SIMPL function calls. What does change radically however, is the blocking behaviour of the two variations. In the original simplistic design the puzzle was submitted via a blocking **Send**. In the reply driven agent design the same puzzle would be submitted via a non-blocking **Reply**. In the original design the puzzle solution was carried in the response message. ie. the GUI would freeze between the puzzle submission and the puzzle solution. With the agent design the puzzle solution will be carried in a blocking **Send** originating at the engine agent side. ie. the GUI would remain unblocked between the submission of the puzzle and the receipt of the solution. The end result would be that no matter how long the engine took to solve a puzzle the GUI would remain responsive throughout.

---

<sup>1</sup>The ARM processor has no such vehicle for "waiting out a child"; a signal called `SA_NOCLDWAIT` must be trapped which allows the kernel to clear zombie children from its process list.

## 18.2 Networked Sudoku

Our original design had the sudoku engine and the Python GUI residing on a single Linux node. One of the advantages of the SIMPL approach is that this can be readily extended to the multi-node case. Our project spec calls for a networked connection between the client GUI and the server engine. Recalling Chapter 7 and relabelling Figure 7.1 we now have Figure 18.2

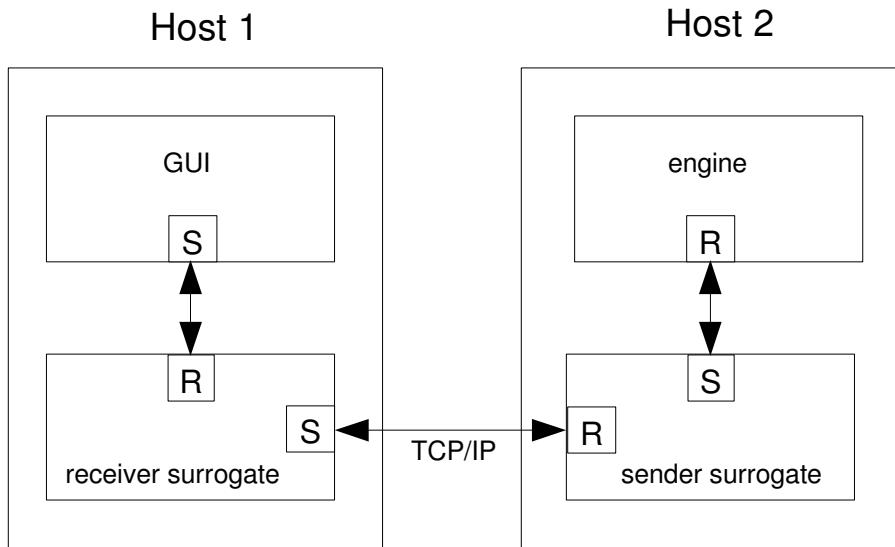


Figure 18.2: Networked Sudoku

We discuss TCP/IP surrogates because the TCP/IP network protocol is deployed widely. Essentially all we have to change to render our code usable across the TCP/IP network is to arrange to have a couple of background daemon processes run on each node and arrange to pass a composite SIMPL name containing a host IP address field into the GUI sender's `name_locate` call.

For simplicity we'll illustrate the networked Sudoku for our *simplest system which would work* initial design: a Python GUI (sender) and a ‘C’ engine (receiver). Embracing the forked engine agent design is equally straightforward. For convenience we will use two start up scripts, one for each host.

On host 1 where the GUI will be running, we have the following start up script (derived in part from our sb004 pretest and dotest scripts):

```
1 # startup script for host 1
2
3#!/bin/sh
4
5export FIFO_PATH=/home/fifo
6export SIMPL_PATH=/home/simpl
7
8 ./stopHost1
9 sleep 1
10
11echo starting protocolRouter
12 $SIMPL_PATH/bin/protocolRouter &
13 sleep 1
14
15echo starting surrogateTcp
16 $SIMPL_PATH/bin/surrogateTcp &
17
18echo ""
19echo "====="
20echo "1) 9x9 puzzle"
21echo "2) 16x16 puzzle"
22echo ""
23echo -n "-> "
24read ans
25
26if [ $ans == '1' ]
27then
28    SIZE=9
29else
30    SIZE=16
31fi
32
33echo starting Python GUI
34 REMOTENAME="SIMPL_TCP:host2:ENGINE"
35 python gui.py gui.py $REMOTENAME $SIZE
```

line 3 Use the default system shell.

line 5 We export the FIFO\_PATH shell variable that is required by SIMPL. Ordinarily, the FIFO\_PATH would be exported at host boot up time via /etc/profile or equivalent.

line 6 We export the SIMPL\_PATH shell variable SIMPL so that we can access the SIMPL surrogate and protocol router required by the script. Again, the

SIMPL\_PATH would also be exported at host boot up time via /etc/profile or equivalent.

line 8 We run a stop script which is in the same directory as this start script in order to shut down any possibly left over programs. The stop script is also available in the source code tarball.

line 12 We start the protocol router program required by the surrogate program in the background.

line 16 We start the surrogate which uses the TCP/IP protocol for remote communications.

lines 18-31 The script asks the user for the size of puzzle to be worked on (9x9 or 16x16).

line 34 **Important:** The SIMPL name of the remote engine program is set here. This is the means by which remote SIMPL communications are achieved. A local communication would merely state the SIMPL name of the receiving program or the the name of the common host and the SIMPL name of the receiving program. Notice that the name is composed of three parts which are delimited by colons. The first field is the protocol to be used, *SIMPL-TCP*. This field entry is actually unnecessary in our case because we have only one type of surrogate running. Nevertheless, it is more general and forces the programmer to be aware that the TCP/IP surrogate has been chosen as the mode of remote SIMPL communications. The second field, *host2*, is the host name of the computer where the Sudoku engine program is running and is naturally unique. Because this host name differs from the host name of the computer running this script, it is understood that remote communications will be required. The last field, *ENGINE*, is the SIMPL name of the Sudoku engine program.

line 35 The Python GUI program is started. The first field is the Python interpreter. It is assumed that there is a *PATH* to this executable. The second field, *gui.py* is the name of the Python GUI program. The third field is the SIMPL name of the Python GUI program. The fourth field is the SIMPL name of the remote Sudoku engine program. Finally, the fifth field indicates the size of the puzzle to be worked on.

On host 2 where the Sudoku engine will be running, we have the following start up script (derived in part from the sb004 dotest script):

```
1 # startup script for host 2
2
3#!/bin/sh
4
5export FIFO_PATH=/home/fifo
6export SIMPL_PATH=/home/simpl
7
8 ./stopHost2
9 sleep 1
10
11echo starting protocolRouter
12 $SIMPL_PATH/bin/protocolRouter &
13 sleep 1
14
15echo starting surrogateTcp
16 $SIMPL_PATH/bin/surrogateTcp &
17
18echo starting logger
19 $SIMPL_PATH/bin/fclogger -n LOGGER >./junk &
20
21echo starting sudoku engine
22 ./engine -n ENGINE -1 LOGGER &
```

**Observe the symmetry between the host 1 and host 2 start up scripts.**

line 3 Use the default system shell.

line 5 We export the FIFO\_PATH shell variable that is required by SIMPL. Ordinarily, the FIFO\_PATH would be exported at host boot up time via /etc/profile or equivalent.

line 6 We export the SIMPL\_PATH shell variable SIMPL so that we can access the SIMPL surrogate and protocol router required by the script. Again, the SIMPL\_PATH would also be exported at host boot up time via /etc/profile or equivalent.

line 8 We run a stop script which is in the same directory as this script in order to shut down any possibly left over programs. The stop script is also available in the source code tarball.

line 12 We start the protocol router program required by the surrogate program.

line 16 We start the surrogate which uses the TCP/IP protocol for remote communications.

line 19 The logger is started. It's a good place to start for debugging problems.

Any problems/warnings/comments etc. will be written to the *junk* file in current directory. As well, don't forget to examine /var/tmp/simpl for warnings/errors relating to problems with SIMPL.

line 22 The Sudoku engine is started up. The SIMPL name of the engine and the name of the logger program follow on the command line.

The startup order is important for SIMPL. A receiver must have registered its SIMPL name before another process tries to connect to it. Hence the startup script for host 2 must be run before the startup script on host 1.<sup>2</sup>

These scripts are modelled after the integration test scripts (sb004) discussed in Chapter 17. The system should run indistinguishably from test sb004. If it does not, clues to problems associated with networked Sudoku will manifest in either the trace log or the main SIMPL log (/var/tmp/simpl on Linux). It is important to note that aside from starting up the relevant surrogate programs, the only difference between sb004 and the networked Sudoku is in the SIMPL name that the Python GUI (sender) uses to locate the Sudoku engine (receiver). On most SIMPL installations, the relevant surrogate and protocol router programs are started and backgrounded during system boot along with other Linux daemon processes.

## 18.3 Tcl/Tk GUI

Python is a very capable language with which to create GUIs. However, SIMPL supports other equally capable languages amongst which is Tcl/Tk.

In the first edition of this book our project design made provision for two GUI's, Python/Tk and Tcl/Tk. The reasons for the two approaches were:

1. Since the Tcl library exists, let's demonstrate its use within the project by way of example.
2. The Python SIMPL library did not support communications via the tclSurrogate. The tclSurrogate was created to remotely exchange SIMPL messages with non-Unix type hosts such as Windows. The Tcl SIMPL library was created for this very purpose. So, if you wanted to run the Sudoku GUI on a Windows host and communicate remotely with a Linux host

---

<sup>2</sup>A more complicated variation of these two scripts can be built to allow for retries to relax this sequencing requirement somewhat.

running the sudoku engine, then it would have to be the Tcl/Tk version of the GUI.

In the second edition of this book number 1 above is still valid, however number 2 is not. Python now has the ability to communicate via the tclSurrogate thus making Windows/non-Windows network communications possible. Moreover, there are now SIMPL libraries for Java as well. Even if the reasoning has changed somewhat, we still regard it as instructive to include the Tcl/Tk version of the Sudoku GUI in some detail.

Depending on when our hypothetical Tcl/Tk GUI team begins work, the engine team may not yet have completed theirs. As such our Tcl/Tk team will begin with a test plan consisting of unit tests sb003 and sb003s, depending on

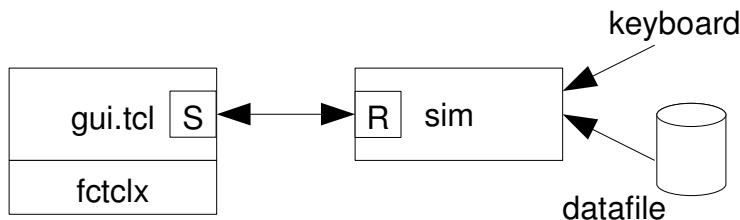


Figure 18.3: sb003 - Tcl GUI unit test

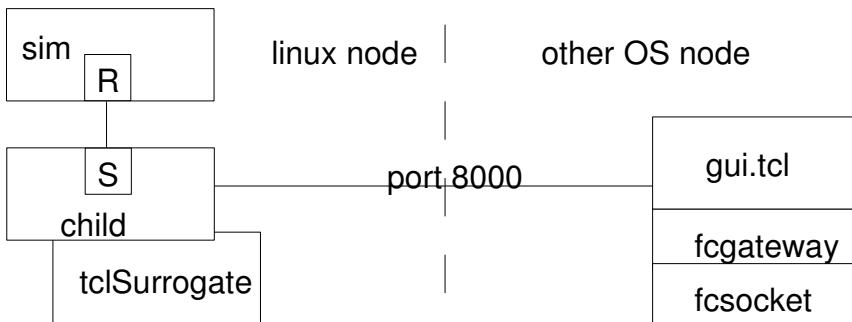


Figure 18.4: sb003s - Non-Linux Tcl GUI unit test

whether the shared guilib or the surrogate guilib is used. The shared guilib would be used for the Linux desktop and the surrogate guilib would be used for non-Linux desktop clients. In both cases the engine simulator would be used in place of the real Sudoku puzzle engine. Once again the module SIMPL design allows

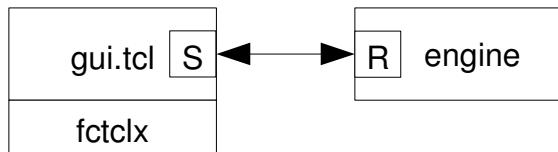


Figure 18.5: sb005 - Tcl GUI/Sudoku engine test

for flexibility in scheduling our hypothetical multi-language developer teams. The integration test is modelled after the similar Python test.

Let's focus on sb003 (see Figure 18.3) because the exercise will be the same for sb003s. What follows is in part very similar to the Python/Tk unit test. The simulator (sim) in this test accesses a hard coded puzzle solution. The GUI contains a hard coded test puzzle which matches this. As such the primary test will consist of pressing the *Test* button in the GUI and verifying the solution that sim replies back to the GUI.

In order to perform our unit test for the Tcl/Tk GUI we run the following commands:

- dotest sb003
- posttest sb003

The pretest script for sb003 will not be needed because all the setup is preset in the GUI code itself and the Tcl/Tk script is only capable of sending 9x9 sized puzzles, unlike the Python/Tk GUI which can also send 16x16 sized puzzles.

The dotest script for sb003 would look something like the version below. Note that the sim and gui.tcl executable files are pulled from the local Sudoku build area. This is standard practice for a unit test.

```

#!/bin/bash

TEST_NO=sb003
TEST_DESC="Tcl/Tk gui unit test"

TEST_DIR=$TEST_HOME/testing/test$TEST_NO
OUTFILE=$TEST_DIR/results/test.out

echo "Starting up test #$TEST_NO" | tee $OUTFILE
echo $TEST_DESC | tee -a $OUTFILE
date | tee -a $OUTFILE

```

```

cd $SIMPL_HOME/bin
fclogger -n LOGGER > $TEST_DIR/results/junk &

cd $TEST_HOME/sudoku/bin
sim -n ENGINE -p $TEST_DIR/data -l LOGGER &

gui.tcl -N TCLGUI -R engine

fcslay ENGINE
fcslay LOGGER

echo "Test finished you can run posttest $TEST_NO for result"

```

There are multiple aspects of the Tcl/Tk GUI that need to be validated. Most of those, such as verifying the *Save* and *Restore* functionality are entirely internal to the client code and don't involve any interaction with the puzzle engine. Test sb003 also concerns itself with unit testing the interaction between the client GUI and the sim. We want to verify the SIMPL message exchange between the client and the sim.

When the *dotest sb003* is run a screen something like Figure 18.6 will come up on the test system (at least after the *Test* button is selected).

The idea is that this matches the canned puzzle in the sim (Sudoku engine stub) and if we submit this we expect to get back a valid solution to this puzzle. Since the puzzle solution will be rendered on the dotest GUI it will be necessary to run the posttest script (which contains the valid solution) from a separate console.

The engine protocol also contains provision for a PUZZLE\_FAILURE token. The sim will respond with this token if any other puzzle than the canned one is submitted. To stimulate this test case we repeat the pretest and dotest steps but rather than selecting the test puzzle we arbitrarily assign a couple of numbers in the blank puzzle and submit that. In this instance the sim is designed to respond with a failure token and we expect that the GUI will present a failure dialog.

All of the code associated with test sb003 is available in the accompanying tarball for the book.<sup>3</sup>

---

<sup>3</sup>If you wish to follow along with some real code in front of you the entire Sudoku project code is available at <http://www.icanprogram.com/simplBook>.

The bulk of the gui.tcl code relates to the widgets presented in the client window itself. These Tcl/Tk details are beyond the scope of this book. Suffice to say that the whole GUI client is contained in less than 350 lines (including white space) of Tcl/Tk script code.

Of more interest to the reader are the SIMPL aspects of the Tcl/Tk GUI script. If we examine the first few lines of the script we find:

```
#!/usr/bin/wish
#
# Tcl/Tk Sudoku Gui
# for SIMPL Book
#
set recvName ENGINE
set MYTOKEN(DO_9)      0
lappend auto_path $env(SIMPL_HOME)/lib
source ./guilib.tcl
```

In the *recvName* variable we are stuffing the default SIMPL name for the sim (a SIMPL receiver). This name will subsequently be used in a **name\_locate** call. We also see the 9x9 puzzle token being captured as a Tcl/Tk array element.

The most important element in this code block is the line which includes the guilib.tcl code. This is the only code which differs between the Linux desktop (shared library) version of our script and the other desktop (surrogate library) version of our script. With the use of Makefile magic we simply arrange to have different lines of code get absorbed into the script depending on our intended desktop. It is worth examining these two variations in more detail. They are contained as the guilib.shared and the guilib.surrogate source files respectively.

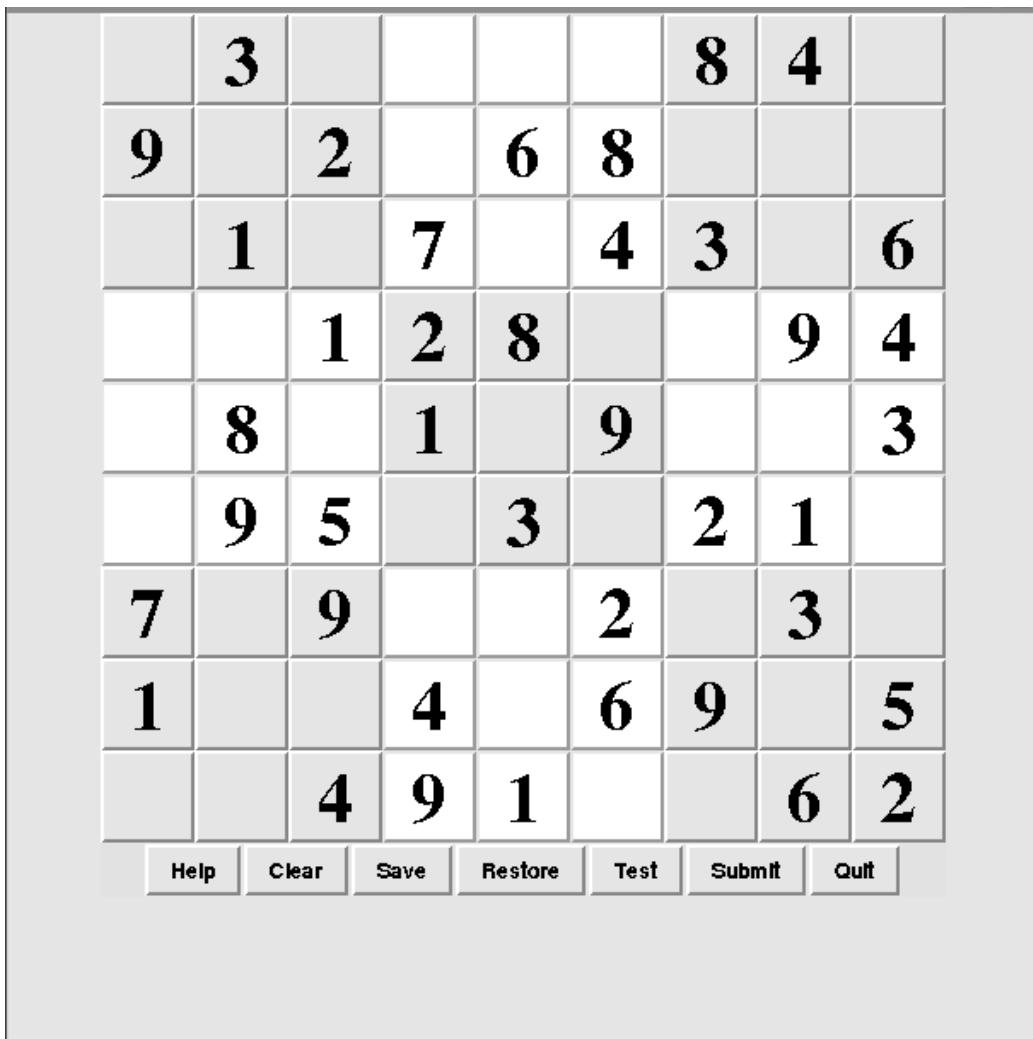


Figure 18.6: Tcl/Tk GUI screen

The contents of guilib.shared are listed below:

```
package require Fctclx
```

Fctclx is the name of the Tcl/Tk SIMPL shared library which extends the SIMPL command set to include all of the basic SIMPL API. viz. ***Send***, ***Receive***, ***Reply*** etc. The guilib.surrogate file is only marginally more complex as shown below:

```
set gatewayAddr 127.0.0.1
set gatewayPort 8000
```

```
package require fcgateway
package require fcsocket
```

Here we need to specify two global variables for the IP address (or URL) and the port number associated with the tclSurrogate partner used to enable transparent SIMPL communications with non-Linux platforms.<sup>4</sup> The fcgateway and the fcsocket are Tcl/Tk packages (libraries) which encapsulate the tclSurrogate protocol (fcsocket) and then abstract this as the SIMPL Send/Receive/Reply API (fcgateway). See Appendix J.<sup>5</sup>

The next SIMPL aspect in our GUI script of note is the section at the bottom which concerns itself with the SIMPL ***name\_attach*** and ***name\_locate*** setup. This is illustrated in the following code snip:

```
set fn main
wm geometry . 600x600+200+0
wm title . "SIMPL Book Tcl/Tk Sudoku Gui"
wm resizable . 0 0

set myName TCLGUI

set state flag
foreach arg $argv {
    switch -- $state {
        flag {
            switch -glob -- $arg {
                -N { set state name }
                -R { set state recv }
                -P { set state port }
                -G { set state gateway }
                default { error "unknown flag $arg" }
            }
        }
        name {
            set myName $arg
            set state flag
        }
        recv {
            set recvName $arg
            set state flag
        }
    }
}
```

---

<sup>4</sup>The sample guilib.surrogate listing shows the loopback address (127.0.0.1) for the gatewayAddr and the default tclSurrogate port (8000) for the gatewayPort.

<sup>5</sup>fcgateway and fcsocket are part of the SIMPL-Tcl/Tk tarball which can be downloaded and installed according to A.6

```

}

port {
    set gatewayPort $arg
    set state flag
}

gateway {
    set gatewayAddr $arg
    set state flag
}
};# end switch state
};# end foreach

name_attach $myName

set recvID [name_locate $recvName]

```

The first part of this logic concerns itself with the capture of command line parameter options associated with gui.tcl. These include things like the -N parameter for overriding the default SIMPL name given to gui.tcl or the -R option for overriding the default SIMPL name for the puzzle engine (sim in this instance) discussed in Chapter 14. This logic is immediately followed by two standard SIMPL code lines for attaching the SIMPL name to the client process and then opening the SIMPL channel to the puzzle solving engine.

There is a ***name\_detach*** section at the end of script which gets executed as a result of selecting the *Quit* button.

```

button $br.quit -text Quit -command {set x 1}
pack $br.quit -side left

pack $br -side bottom
pack $f

vwait x

name_detach

exit

```

The vwait command in Tcl waits on a change in a variable while continuing to execute the Tk event loop. When we select *Quit* we change this monitored variable which sends the script to the exit and passing the ***name\_detach*** call en route.

The final SIMPL code block of interest is contained in the hndlSubmit procedure reproduced below:

```

#=====
# hndlSubmit - entry point
#=====
proc hndlSubmit {} {
    global recvID
    global MYTOKEN
    global puzzle

    hndlSave

    set elements ""
    for {set r 0} {$r < 9} {incr r} {
        for {set c 0} {$c < 9} {incr c} {
            set elements [format "%s%s" $elements $puzzle(.f.$r.$c)]
        };# end for c
    };#end for r

    set rc -1
    if {$recvID != -1} {
        set sMsg [binary format "i1a*" \
            $MYTOKEN(DO_9) \
            $elements]

        set sBytes [string length $sMsg]
        set rMsg [Send $recvID $sMsg $sBytes]

        binary scan $rMsg i1i1i1a* slot rbytes token rpuzzle

        if {$token != $MYTOKEN(DO_9)} {
            tk_messageBox -message [format "puzzle error"]
        } else {

            set i 0
            for {set r 0} {$r < 9} {incr r} {
                for {set c 0} {$c < 9} {incr c} {
                    set puzzle(.f.$r.$c) [string index $rpuzzle $i]
                    incr i
                };#end for c
            };# end for r
            set rc 1
        };# end else token
    };# end if recvID

    return $rc
}

```

```
};# end hndlSubmit
```

One can easily spot the SIMPL ***Send*** command and its tokenized message setup preamble including the assignment of the token for a 9x9 puzzle. The binary scan command which follows is decoding the reply which comes back from the puzzle engine. The Tcl/Tk libraries prepend some header information (slot, rbytes) to the actual SIMPL message.

The rest of the gui.tcl script concerns itself with widget manipulation to provide the balance of the functionality in our Sudoku puzzle client.

## 18.4 Java/Swing GUI

Another powerful language for creating GUIs is Java. Since the first edition of this book was published SIMPL has grown to embrace the Java language. Java can now be used to create SIMPL modules including a Java/Swing Sudoku GUI. Like the other GUI languages which SIMPL supports, Java has two libraries: Jsimpl and Ssimpl. Jsimpl can be used by Java programs to SIMPL communicate locally or via the surrogates on a network of Linux hosts. Ssimpl can be used if communications are such that the GUI is running on a non-Linux (Windows) host and the engine is running on a Linux host.

For this discussion we suppose that our hypothetical Java team begins work after the engine team has a reasonably function prototype available. The Java team has a testplan modelled on the Python integration test:

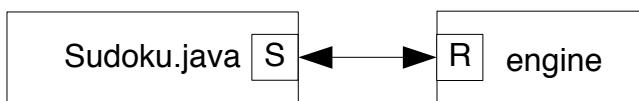


Figure 18.7: sb007 - Java/Swing test

The sb007 runtest script looks something like:

*sudoku\_java*

```

1
2 #!/bin/bash
3
4 TEST_NO=sb007
5 TEST_DESC="integration test with Java/Swing gui"
6
  
```

```
7 TEST_DIR=$TEST_HOME/testing/test$TEST_NO
8 OUTFILE=$TEST_DIR/results/test.out
9
10 echo "Starting up test #$TEST_NO" | tee $OUTFILE
11 echo $TEST_DESC | tee -a $OUTFILE
12 date | tee -a $OUTFILE
13
14 cd $SIMPL_HOME/bin
15 fclogger -n LOGGER > $TEST_DIR/results/junk &
16
17 cd $TEST_HOME/sudoku/bin
18 cksum engine | tee -a $OUTFILE
19 engine -n ENGINE -l LOGGER &
20 sleep 1
21
22 CLASSPATH=../class:$SIMPL_HOME/java/class
23 LIBPATH=../lib:$SIMPL_HOME/java/lib
24
25 java -classpath $CLASSPATH -Djava.library.path=$LIBPATH Sudoku
26
27 fcslay ENGINE
28 fcslay LOGGER
29
30 echo "Test finished you can run posttest $TEST_NO for result"
```

The important features of the script are as follows:

line 3 This is where Java finds any required class files.

line 5 If the program Sudoku.java is set to run an instance of Jsimpl then this line is commented out. If it is set to run an instance of Ssimpl then the tclSurrogate is required.

line 9 Invocation of the Java program - Sudoku.class.

line 12 Required only if running tclSurrogate.

The Java GUI code looks something like:

### Sudoku.java

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
```

```
4  class Sudoku extends JFrame
5  {
6      private static final long serialVersionUID = 42L;
7      private static final int SIZE = 81;
8      private static final int DO_9 = 0;
9      private JButton key []=new JButton [SIZE];
10     private String numbers []={” ”,”1”,”2”,”3”,”4”,”5”,”6”,”7”,”8”,”9” };
11     private String saved []=new String [SIZE];
12     private static int receiver;
13     /*
14      Either one of the following will work depending on the situation
15     */
16     Jsimpl simpl = new Jsimpl();
17     //Simpl simpl = new Simpl();
18
19     public static void main( String [] args)
20     {
21         Sudoku win = new Sudoku();
22         win .setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
23         win .setVisible (true);
24     }
25
26
27     public Sudoku()
28     {
29         int ret;
30
31         ret = simpl.nameAttach(”javaSudoku” );
32         //ret = simpl.nameAttach(50000, ”localhost”, ”javaSudoku” );
33         if (ret == -1)
34         {
35             System.out.println(”name attach error” );
36             System.exit (-1);
37         }
38
39         receiver = simpl.nameLocate(”ENGINE” );
40         if (receiver == -1)
41         {
42             System.out.println(”name locate error” );
43             System.exit (-1);
44         }
45
46         // create matrix elements and panel
47         JPanel elements = new JPanel();
48         elements.setLayout(new GridLayout(9, 9, 1, 1));
49         ActionListener numListener = new NumListener();
```

```
50  for (int i=0; i < SIZE; i++)
51  {
52      key[ i ] = new JButton();
53      key[ i ].setFont(new Font("Helvetica", Font.BOLD, 25));
54      key[ i ].setText(" ");
55      elements.add(key[ i ]);
56      key[ i ].addActionListener(numListener);
57  }
58
59 // create command buttons panel
60 JPanel command = new JPanel();
61 command.setLayout(new GridLayout(1, 5, 5, 0));
62
63 JButton clearButton = new JButton("Clear");
64 clearButton.setFont(new Font("Helvetica", Font.BOLD, 15));
65 command.add(clearButton);
66 ActionListener clearListener = new ClearListener();
67 clearButton.addActionListener(clearListener);
68
69 JButton saveButton = new JButton("Save");
70 saveButton.setFont(new Font("Helvetica", Font.BOLD, 15));
71 command.add(saveButton);
72 ActionListener saveListener = new SaveListener();
73 saveButton.addActionListener(saveListener);
74
75 JButton restoreButton = new JButton("Restore");
76 restoreButton.setFont(new Font("Helvetica", Font.BOLD, 15));
77 command.add(restoreButton);
78 ActionListener restoreListener = new RestoreListener();
79 restoreButton.addActionListener(restoreListener);
80
81 JButton testButton = new JButton("Test");
82 testButton.setFont(new Font("Helvetica", Font.BOLD, 15));
83 command.add(testButton);
84 ActionListener testListener = new TestListener();
85 testButton.addActionListener(testListener);
86
87 JButton submitButton = new JButton("Submit");
88 submitButton.setFont(new Font("Helvetica", Font.BOLD, 15));
89 command.add(submitButton);
90 ActionListener submitListener = new SubmitListener();
91 submitButton.addActionListener(submitListener);
92
93 // top level content panel
94 JPanel content = new JPanel();
95 content.setLayout(new BorderLayout(0, 6));
```

```
96     content.add(elements, BorderLayout.CENTER);
97     content.add(command, BorderLayout.SOUTH );
98     content.setBorder(BorderFactory.createEmptyBorder(3,3,3,3));
99
100    // finish the window
101    this.setContentPane(content);
102    this.pack();
103    this.setTitle("SIMPL Sudoku 9x9 in Java");
104    this.setResizable(false);
105    this.setLocationRelativeTo(null);
106    setSize(500, 550);
107 }
108
109 class NumListener implements ActionListener
110 {
111     public void actionPerformed(ActionEvent event)
112     {
113         // get/set button text
114         JButton b = (JButton)event.getSource();
115         String digit = b.getText();
116
117         for(int j = 0; j < 10; j++)
118         {
119             if (digit.equals(numbers[j]))
120             {
121                 if (digit == "9")
122                 {
123                     digit = numbers[0];
124                 }
125                 else
126                 {
127                     digit = numbers[j+1];
128                 }
129                 break;
130             }
131         }
132         b.setText(digit);
133     }
134 }
135
136 class ClearListener implements ActionListener
137 {
138     public void actionPerformed(ActionEvent event)
139     {
140         for (int i=0; i < key.length; i++)
141         {
```

```
142         key[ i ].setText(" ");
143     }
144   }
145 }
146
147 class SaveListener implements ActionListener
148 {
149   public void actionPerformed(ActionEvent event)
150   {
151     for (int i=0; i < SIZE; i++)
152     {
153       saved[ i ] = key[ i ].getText();
154     }
155   }
156 }
157
158 class RestoreListener implements ActionListener
159 {
160   public void actionPerformed(ActionEvent event)
161   {
162     for (int i=0; i < SIZE; i++)
163     {
164       key[ i ].setText(saved[ i ]);
165     }
166   }
167 }
168
169 class TestListener implements ActionListener
170 {
171   public void actionPerformed(ActionEvent event)
172   {
173     for (int i=0; i < SIZE; i++)
174     {
175       key[ i ].setText(" ");
176     }
177
178     key[ 3 ].setText("7");
179     key[ 6 ].setText("1");
180     key[ 10 ].setText("8");
181     key[ 11 ].setText("1");
182     key[ 12 ].setText("4");
183     key[ 17 ].setText("6");
184     key[ 22 ].setText("1");
185     key[ 24 ].setText("3");
186     key[ 25 ].setText("4");
187     key[ 27 ].setText("1");
```

```
188     key[29].setText("9");
189     key[34].setText("5");
190     key[39].setText("6");
191     key[41].setText("7");
192     key[46].setText("6");
193     key[51].setText("4");
194     key[53].setText("3");
195     key[55].setText("3");
196     key[56].setText("4");
197     key[58].setText("2");
198     key[63].setText("9");
199     key[68].setText("6");
200     key[69].setText("8");
201     key[70].setText("2");
202     key[74].setText("2");
203     key[77].setText("5");
204   }
205 }
206
207 class SubmitListener implements ActionListener
208 {
209   public void actionPerformed(ActionEvent event)
210   {
211     int outToken = DO_9;
212     int inToken = -1;
213     int ret = -1;
214     int i;
215     byte[] matrixElements = new byte[SIZE];
216
217     // check for values
218     for (i=0; i < SIZE; i++)
219     {
220       if (key[i].getText() != " ")
221       {
222         break;
223       }
224     }
225     if (i == SIZE)
226     {
227       return;
228     }
229
230     // save matrix elements
231     for (i=0; i < SIZE; i++)
232     {
233       String str = key[i].getText();
```

```
234     saved[ i ] = str;  
235  
236     // java uses strings here, need to send single bytes  
237     if (str == " ")  
238     {  
239         matrixElements[ i ] = 32;  
240     }  
241     else if (str == "1")  
242     {  
243         matrixElements[ i ] = 49;  
244     }  
245     else if (str == "2")  
246     {  
247         matrixElements[ i ] = 50;  
248     }  
249     else if (str == "3")  
250     {  
251         matrixElements[ i ] = 51;  
252     }  
253     else if (str == "4")  
254     {  
255         matrixElements[ i ] = 52;  
256     }  
257     else if (str == "5")  
258     {  
259         matrixElements[ i ] = 53;  
260     }  
261     else if (str == "6")  
262     {  
263         matrixElements[ i ] = 54;  
264     }  
265     else if (str == "7")  
266     {  
267         matrixElements[ i ] = 55;  
268     }  
269     else if (str == "8")  
270     {  
271         matrixElements[ i ] = 56;  
272     }  
273     else  
274     {  
275         matrixElements[ i ] = 57;  
276     }  
277 }  
278  
279 // reset the message buffers
```

```
280     simpl.reset();  
281  
282     // pack the message token  
283     ret = simpl.packInt(outToken, simpl.SIMPL_BIN);  
284     if (ret == -1)  
285     {  
286         System.out.println("Problem with packing the token\n");  
287     }  
288  
289     // pack the matrix elements  
290     ret = simpl.packByteArray(matrixElements, simpl.SIMPL_BIN);  
291     if (ret == -1)  
292     {  
293         System.out.println("Problem with packing the elements\n");  
294     }  
295  
296     // send the message to the sudoku engine  
297     ret = simpl.Send(receiver);  
298     if (ret == -1)  
299     {  
300         System.out.println("send error");  
301         System.exit(-1);  
302     }  
303  
304     // get the incoming message token  
305     inToken = simpl.unpackInt(simpl.SIMPL_BIN);  
306     // get the solution  
307     ret = simpl.unpackByteArray(matrixElements, simpl.SIMPL_BIN);  
308  
309     // not a successful solution?  
310     if (inToken != outToken)  
311     {  
312         System.out.println("Problem with making a solution\n");  
313     }  
314     else  
315     {  
316         // java wants strings, bytes have been returned  
317         for (i=0; i < SIZE; i++)  
318         {  
319             switch(matrixElements[i])  
320             {  
321                 case 49:  
322                     key[i].setText("1");  
323                     break;  
324                 case 50:  
325                     key[i].setText("2");  
326             }  
327         }  
328     }  
329 }
```

```
326         break;
327     case 51:
328         key[ i ].setText( "3" );
329         break;
330     case 52:
331         key[ i ].setText( "4" );
332         break;
333     case 53:
334         key[ i ].setText( "5" );
335         break;
336     case 54:
337         key[ i ].setText( "6" );
338         break;
339     case 55:
340         key[ i ].setText( "7" );
341         break;
342     case 56:
343         key[ i ].setText( "8" );
344         break;
345     case 57:
346         key[ i ].setText( "9" );
347         break;
348     default:
349         key[ i ].setText( " " );
350     }
351   }
352 }
353 }
354 }
355 }
```

The features of the Java GUI program that are important to SIMPL are as follows:

line 17 An instance of Jsimpl is created. This would be the case if all hosts involved are non-Windows, either locally or remotely.

line 18 An instance of Ssimpl is created. This would be the case if the host running Sudoku.class is Windows based and the engine program is running on a remote non-Windows host. This would require the use of the tclSurrogate. Note the script sudoku\_java outlined above.

line 31 One can infer from this line that this is a Jsimpl instance and that the GUI and engine programs are both running on the same non-Windows host.

line 32 One can infer from this line that this is a Ssimpl instance and that the GUI is running on a Windows host while the engine programs is running on a remote non-Windows host.

lines 280-294 The SIMPL message to the engine is constructed via the pack methods.

lines 297-302 The SIMPL message is sent to the engine program.

lines 305-307 The reply message from the engine program is deconstructed via the unpack methods.

When sb007 is run this GUI interface is produced:

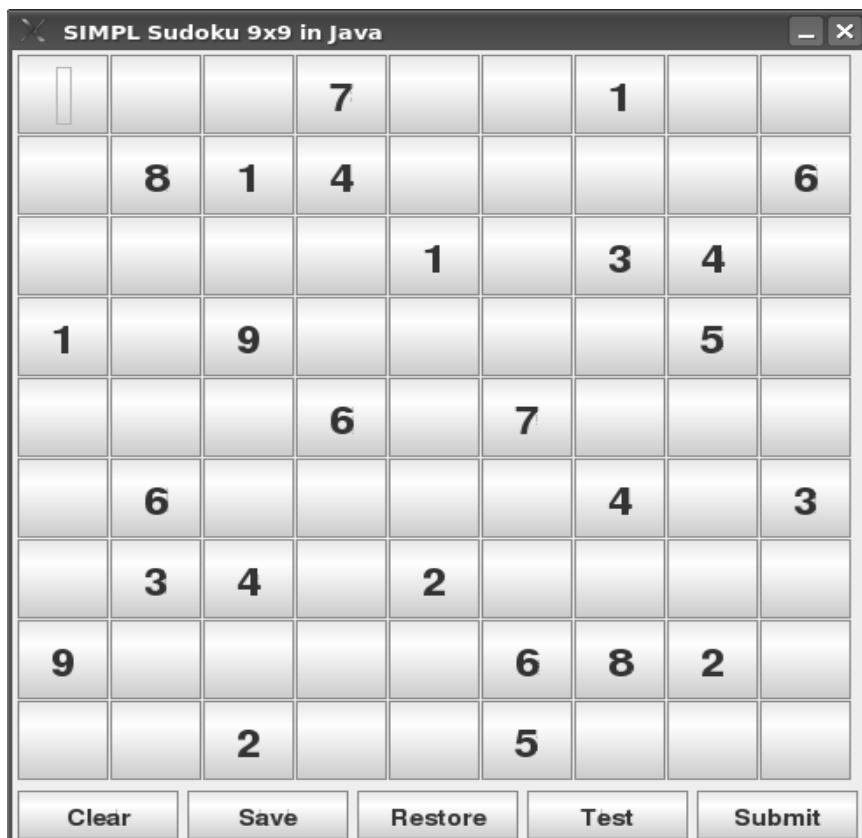


Figure 18.8: Java/Swing GUI screen

## 18.5 CGI

So far our GUI evolution has been in the direction of expanding the language choice to include Python, Tcl, and Java. However, in all these cases the user would need to install the respective runtime enabling libraries alongside the GUI code itself in order to be able to solve Sudoku puzzles.

Another option discussed in our original project spec allowed for Sudoku puzzles to be submitted via a web browser. Web browsers are very pervasive and most allow for the same basic methodologies such as CGI (Common Gateway Interface). This approach to application design has become known as web services. The SIMPL-CGI approach can be thought of as web services lite.

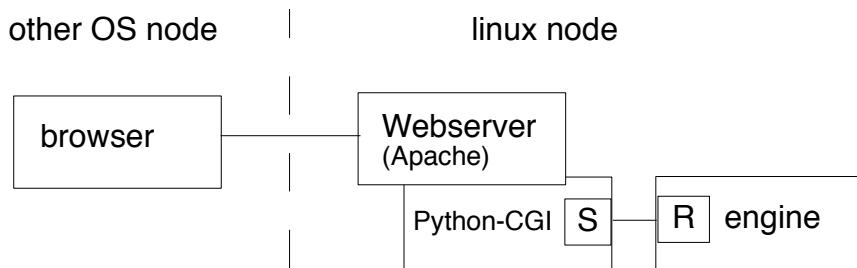


Figure 18.9: sb006 - Python-CGI test

CGI is a mechanism allowing a web server to invoke a program, capture the output of that program, wrap it in an HTTP header and return it to a web browser.

SIMPL function calls have been used in CGI programs. However, there are a number of issues that have to be addressed before this can happen. We will proceed through them one step at a time.

### 18.5.1 Web Server

We are going to use Apache as our example of a web server. We have chosen Apache specifically due to its widespread use. Moreover, the following discourse regarding Apache assumes that we are using a Linux host as the server. Apache spawns the CGI program and passes data to that program. The data usually originates from an HTML *form* of some sort that has been filled in via a web browser. The CGI program passes its processed data back to Apache via its STDOUT. Essentially, the order of events is as follows:

1. A browser connects to a server.
2. A form of some sort is sent to the browser from the server.
3. The form data is filled in on the browser and sent back to the server.
4. Upon receiving the completed form, the server spawns the appropriate CGI program.
5. The data from the form is extracted by the CGI program.
6. The data from the form is processed by the CGI program. In our case, the CGI program is SIMPL sending the contents of the form to the Sudoku puzzle solving program. The solution of the puzzle is then replied back to the CGI program from the puzzle solving program.
7. The results of the processing are returned to the browser by the CGI program via the server.

In order for Apache to know how to proceed, it may be necessary to modify the Apache configuration file. Required directives are covered in Appendix L.

### 18.5.2 Sudoku via Web Browser

We want to have Sudoku puzzle values entered via a browser and when solved, have the solution also displayed by the browser. Note that for our example we will run everything on one host computer, ordinarily this would not be the case. We will discuss this more fully at the end. To recap and expand, the situation is as follows:

1. The Apache web server is started on the local host. We recommend using something like:

**apachectl start**

because apachectl looks after a mess of details for you.

2. The sudoku puzzle engine program is started in the background on the local host:

**engine -n ENGINE &**

3. The browser is brought up and a URL of `http://localhost/sudoku.html` is entered. By entering `localhost` (127.0.0.1) in the URL you are using the loopback capability on your computer. The HTML file `sudoku.html` contains a form which resembles a sudoku puzzle.
4. When the form values are filled in, the form is sent to the server. The server spawns the CGI python script called `sudoku-cgi.py`.
5. `sudoku-cgi.py` strips out the form values (sudoku puzzle clues) and SIMPL sends them to the puzzle solving engine program for possible solution.
6. The puzzle solving engine replies the sudoku results to `sudoku-cgi.py`.
7. `sudoku-cgi.py` returns an HTML message back to the browser via the server with the results.

### 18.5.3 HTML/CSS

The following HTML file is rendered by the browser and serves as the entry point for 9x9 Sudoku puzzles. As a rule, the CSS components are kept separate in their own file but for the sake of brevity and completeness, they have been included within this file (in the element `<style>`).

#### *sudoku.html*

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Sudoku 9x9 Puzzle</title>
5     <style type="text/css">
6       h1
7       {
8         text-align: center;
9       }
10      body
11      {
12        width: 900px;
13        height: 900px;
14        background-repeat: repeat;
15        background-attachment: scroll;
16        background-position: center center;
17      }
18
19      table
20      {
21        border: 5px solid black;
22        border-collapse: collapse;
23        background: black;
24      }
```

```
25     margin-top: 30px;
26 }
27
28 input
29 {
30   border: 1px solid black;
31   width: 1.75em;
32   height: 1.75em;
33   text-align: center;
34   font-size: 1.5em ;
35   font-weight: 900;
36   background: #FFFF00;
37 }
38
39 .numA
40 {
41   border-right-width: 4px;
42   border-left-width: 1px;
43   border-top-width: 1px;
44   border-bottom-width: 1px;
45 }
46
47 .numB
48 {
49   border-right-width: 1px;
50   border-left-width: 1px;
51   border-top-width: 1px;
52   border-bottom-width: 4px;
53 }
54
55 .numC
56 {
57   border-right-width: 4px;
58   border-left-width: 1px;
59   border-top-width: 1px;
60   border-bottom-width: 4px;
61 }
62
63 input:hover , input:focus
64 {
65   background-color: #ff0000 ;
66 }
67
68 #submit , #reset
69 {
70   border: 2px solid black;
71   width: 10em;
72   height: 1.5em;
73   text-align: center;
74   font: 1.25em bold;
75   background: gray;
76 }
77
78 a:link
79 {
80   color: #3333FF;
81   padding-left: 5px;
82   padding-right: 5px;
83 }
```

```
84     </style>
85   </head>
86
87   <body>
88     <h1>Welcome to the SIMPL Sudoku Game!</h1>
89     <form method="post" action="http://localhost/cgi-bin/sudoku-cgi.py">
90       <div align="center">
91         <h3>Sudoku 9x9 Puzzle Solver</h3>
92         <h4>Click on each box and enter a number 1-9</h4>
93
94       <table summary="numbers from 1-9">
95         <tbody>
96           <tr> <!-- First Row -->
97             <td><input name="c1" type="text" maxlength="1" size="1"/></td>
98             <td><input name="c2" type="text" maxlength="1" size="1"/></td>
99             <td><input name="c3" class="numA" type="text" maxlength="1"
100            size="1"/></td>
101            <td><input name="c4" type="text" maxlength="1" size="1"/></td>
102            <td><input name="c5" type="text" maxlength="1" size="1"/></td>
103            <td><input name="c6" class="numA" type="text" maxlength="1" size="1"
104            value="2"/></td>
105            <td><input name="c7" type="text" maxlength="1" size="1" value="7"/></td>
106            <td><input name="c8" type="text" maxlength="1" size="1"/></td>
107            <td><input name="c9" type="text" maxlength="1" size="1"/></td>
108          </tr>
109
110          <tr> <!-- Second Row -->
111            <td><input name="c10" type="text" maxlength="1" size="1"/></td>
112            <td><input name="c11" type="text" maxlength="1" size="1"
113            value="4"/></td>
114            <td><input name="c12" class="numA" type="text" maxlength="1"
115            size="1"/></td>
116            <td><input name="c13" type="text" maxlength="1" size="1"/></td>
117            <td><input name="c14" type="text" maxlength="1" size="1"/></td>
118            <td><input name="c15" class="numA" type="text" maxlength="1"
119            size="1"/></td>
120            <td><input name="c16" type="text" maxlength="1" size="1"/></td>
121            <td><input name="c17" type="text" maxlength="1" size="1"
122            value="3"/></td>
123            <td><input name="c18" type="text" maxlength="1" size="1"
124            value="5"/></td>
125          </tr>
126
127          <tr> <!-- Third Row -->
128            <td><input name="c19" class="numB" type="text" maxlength="1"
129            size="1"/></td>
130            <td><input name="c20" class="numB" type="text" maxlength="1" size="1"
131            value="1"/></td>
132            <td><input name="c21" class="numC" type="text" maxlength="1" size="1"
133            value="7"/></td>
134            <td><input name="c22" class="numB" type="text" maxlength="1" size="1"
135            value="3"/></td>
136            <td><input name="c23" class="numB" type="text" maxlength="1"
137            size="1"/></td>
138            <td><input name="c24" class="numC" type="text" maxlength="1" size="1"
139            value="6"/></td>
140            <td><input name="c25" class="numB" type="text" maxlength="1"
141            size="1"/></td>
142            <td><input name="c26" class="numB" type="text" maxlength="1"
```

```
143 | size="1"/></td>
144 |     <td><input name="c27" class="numB" type="text" maxlength="1"
145 |     size="1"/></td>
146 |     </tr>
147 |
148 |     <tr> <!-- Fourth Row -->
149 |         <td><input name="c28" type="text" maxlength="1" size="1"/></td>
150 |         <td><input name="c29" type="text" maxlength="1" size="1"/></td>
151 |         <td><input name="c30" class="numA" type="text" maxlength="1" size="1"
152 | value="5"/></td>
153 |         <td><input name="c31" type="text" maxlength="1" size="1"
154 | value="7"/></td>
155 |         <td><input name="c32" type="text" maxlength="1" size="1"/></td>
156 |         <td><input name="c33" class="numA" type="text" maxlength="1"
157 | size="1"/></td>
158 |         <td><input name="c34" type="text" maxlength="1" size="1"/></td>
159 |         <td><input name="c35" type="text" maxlength="1" size="1"
160 | value="1"/></td>
161 |         <td><input name="c36" type="text" maxlength="1" size="1"
162 | value="4"/></td>
163 |     </tr>
164 |
165 |     <tr> <!-- Fifth Row -->
166 |         <td><input name="c37" type="text" maxlength="1" size="1"
167 | value="9"/></td>
168 |         <td><input name="c38" type="text" maxlength="1" size="1"
169 | value="8"/></td>
170 |         <td><input name="c39" class="numA" type="text" maxlength="1"
171 | size="1"/></td>
172 |         <td><input name="c40" type="text" maxlength="1" size="1"/></td>
173 |         <td><input name="c41" type="text" maxlength="1" size="1"/></td>
174 |         <td><input name="c42" class="numA" type="text" maxlength="1"
175 | size="1"/></td>
176 |         <td><input name="c43" type="text" maxlength="1" size="1"/></td>
177 |         <td><input name="c44" type="text" maxlength="1" size="1"
178 | value="7"/></td>
179 |         <td><input name="c45" type="text" maxlength="1" size="1"
180 | value="6"/></td>
181 |     </tr>
182 |
183 |     <tr> <!-- Sixth Row -->
184 |         <td><input name="c46" class="numB" type="text" maxlength="1" size="1"
185 | value="1"/></td>
186 |         <td><input name="c47" class="numB" type="text" maxlength="1" size="1"
187 | value="7"/></td>
188 |         <td><input name="c48" class="numC" type="text" maxlength="1"
189 | size="1"/></td>
190 |         <td><input name="c49" class="numB" type="text" maxlength="1"
191 | size="1"/></td>
192 |         <td><input name="c50" class="numB" type="text" maxlength="1"
193 | size="1"/></td>
194 |         <td><input name="c51" class="numC" type="text" maxlength="1" size="1"
195 | value="4"/></td>
196 |         <td><input name="c52" class="numB" type="text" maxlength="1" size="1"
197 | value="3"/></td>
198 |         <td><input name="c53" class="numB" type="text" maxlength="1"
199 | size="1"/></td>
200 |         <td><input name="c54" class="numB" type="text" maxlength="1"
201 | size="1"/></td>
```

```

202     </tr>
203
204     <tr> <!-- Seventh Row -->
205         <td><input name="c55" type="text" maxlength="1" size="1"/></td>
206         <td><input name="c56" type="text" maxlength="1" size="1"/></td>
207         <td><input name="c57" class="numA" type="text" maxlength="1"
208             size="1"/></td>
209         <td><input name="c58" type="text" maxlength="1" size="1"
210             value="6"/></td>
211         <td><input name="c59" type="text" maxlength="1" size="1"/></td>
212         <td><input name="c60" class="numA" type="text" maxlength="1" size="1"
213             value="7"/></td>
214         <td><input name="c61" type="text" maxlength="1" size="1"
215             value="8"/></td>
216         <td><input name="c62" type="text" maxlength="1" size="1"
217             value="2"/></td>
218         <td><input name="c63" type="text" maxlength="1" size="1"/></td>
219     </tr>
220
221     <tr> <!-- Eighth Row -->
222         <td><input name="c64" type="text" maxlength="1" size="1"
223             value="3"/></td>
224         <td><input name="c65" type="text" maxlength="1" size="1"
225             value="2"/></td>
226         <td><input name="c66" class="numA" type="text" maxlength="1"
227             size="1"/></td>
228         <td><input name="c67" type="text" maxlength="1" size="1"/></td>
229         <td><input name="c68" type="text" maxlength="1" size="1"/></td>
230         <td><input name="c69" class="numA" type="text" maxlength="1"
231             size="1"/></td>
232         <td><input name="c70" type="text" maxlength="1" size="1"/></td>
233         <td><input name="c71" type="text" maxlength="1" size="1"
234             value="9"/></td>
235         <td><input name="c72" type="text" maxlength="1" size="1"/></td>
236     </tr>
237
238     <tr> <!-- Ninth Row -->
239         <td><input name="c73" type="text" maxlength="1" size="1"/></td>
240         <td><input name="c74" type="text" maxlength="1" size="1"/></td>
241         <td><input name="c75" class="numA" type="text" maxlength="1" size="1"
242             value="8"/></td>
243         <td><input name="c76" type="text" maxlength="1" size="1"
244             value="2"/></td>
245         <td><input name="c77" type="text" maxlength="1" size="1"/></td>
246         <td><input name="c78" class="numA" type="text" maxlength="1"
247             size="1"/></td>
248         <td><input name="c79" type="text" maxlength="1" size="1"/></td>
249         <td><input name="c80" type="text" maxlength="1" size="1"/></td>
250         <td><input name="c81" type="text" maxlength="1" size="1"/></td>
251     </tr>
252     </tbody>
253 </table>
254 <p><input id="submit" type="submit" value="Get Solution" />
255 <input id="reset" type="reset" value="Start Over" /></p>
256 </div>
257 </form>
258 </body>
259 </html>

```

### Sudoku 9x9 Puzzle Solver

Click on each box and enter a number 1-9

						2	7	
	4						3	5
	1	7	3		6			
		5	7				1	4
9	8						7	6
1	7				4	3		
			6		7	8	2	
3	2						9	
		8	2					

Figure 18.10: HTML GUI screen

The one feature of this HTML/CSS script that is important to SIMPL is as follows: Line 89 indicates the action to be taken by the server upon receipt of this form. Note that the Python script `sudoku-cgi.py` located in `localhost/cgi-bin` is to handle the form.

#### 18.5.4 CGI Script

The following Python script serves as the CGI program that acts as intermediary between the HTML form sent by the browser and the sudoku engine program.

sudoku-cgi.py

```
1 #!/usr/bin/python3
2
3 # required modules
4 import csimpl
5 import cgi
6
7 # cgi script testing bits
8 """
9 import cgitb
10 cgitb.enable(display=0, logdir="/var/tmp")
11 cgi.test()
12 """
13
14 # set some 'constants'
15 HTML = 4
16 SIZE = 9
17 DO_9 = 0
18 DO_16 = 1
19 BAD_ENTRY = 2
20 SOLUTION_FAILURE = 3
21 FORK_ERROR = 4
22 TOO_MANY_ERROR = 5
23 TIME_ERROR = 6
24 BAD_TOKEN = 7
25 NAME_LOCATE_FAILURE = 8
26 SEND_FAILURE = 9
27
28 # global variables
29 global entries
30 global results
31
32 ****
33 # send back any error information to the browser for viewing
34 def problem(errStr):
35     # must start with a blank line
36     print("Content-type: text/html\n")
37     # DOCTYPE
38     print("<!DOCTYPE html>\n")
39     print("<html lang=\"en\">\n")
40     # head
41     print("<head><title>Sudoku Puzzle</title></head>\n")
42     #body
43     print("<body> <h1><center>%s</center></h1> </body> </html>\n" %(errStr))
44
45 ****
46 # send puzzle entries to the solver engine and hopefully get a solution back
47 def sendPuzzleToEngine():
48     # make an instance of Simpl
49     puzz = csimpl.Simpl("SUDOKU", 1024)
50
51     # name locate the program "ENGINE"
52     receiverId = puzz.nameLocate("ENGINE")
53     if receiverId == -1:
54         return NAME_LOCATE_FAILURE
55
56     # make the message to send
57     puzz.packMsg(puzz.SIMPL_BIN, "iis", DO_9, HTML, entries)
```

```
58
59     # send the message
60     retVal = puzz.send(receiverId)
61     if retVal == -1:
62         return SEND_FAILURE
63
64     # process the reply message; the source field is not used
65     global results
66     token, source, results = puzz.unpackMsg(puzz.SIMPL_BIN, "iis1s")
67
68     return token
69
70 #####
71 # send results to the browser for display
72 def sendResultsToBrowser():
73     # the puzzle cell numbers with special bordering needs
74     a = (2, 5, 11, 14, 29, 32, 38, 41, 56, 59, 65, 68, 74, 77)
75     b = (18, 19, 21, 22, 24, 25, 26, 45, 46, 48, 49, 51, 52, 53)
76     c = (20, 23, 47, 50)
77
78     # must start with a blank line
79     print("Content-type: text/html\n")
80     # DOCTYPE
81     print("<!DOCTYPE html>\n")
82     print("<html lang=\"en\">\n")
83     # head; css styling
84     print("<head><style type=\"text/css\">\n")
85     print("table { border: 5px solid black; border-collapse: collapse; \
86           background: black; margin-top: 30px; }\n")
87     print("input { border: 1px solid black; width: 1.75em; height: 1.75em; \
88           text-align: center; font-size: 1.5em; font-weight: 900; \
89           background: #FFFF00; foreground=black; font:black }\n")
90     print(".numA { border-right-width: 4px; border-left-width: 1px; \
91           border-top-width: 1px; border-bottom-width: 1px; }\n")
92     print(".numB { border-right-width: 1px; border-left-width: 1px; \
93           border-top-width: 1px; border-bottom-width: 4px; }\n")
94     print(".numC { border-right-width: 4px; border-left-width: 1px; \
95           border-top-width: 1px; border-bottom-width: 4px; }\n")
96     print("</style> <title>Sudoku Puzzle Solution</title> </head>\n")
97     # body
98     print("<body> <div align=\"center\"> <h1>Puzzle Solution</h1> <table> \
99           <tbody>\n")
100    # make rows and columns and populate with resultant numbers
101    for i in range(SIZE * SIZE):
102        if not i % SIZE:
103            print("<tr>\n")
104            name = "c" + str(i+1)
105            if i in a:
106                iam = "numA"
107            elif i in b:
108                iam = "numB"
109            elif i in c:
110                iam = "numC"
111            else:
112                iam = ""
113            print("<td><input name=%s class=%s type=\"text\" \
114               maxlength=\"1\" size=\"1\" value=%s></td>\n" \
115               % (name, iam, chr(results[i])))
116        if not (i+1) % SIZE:
```

```

117     print("</tr>\n")
118     print("</tbody> </table> <p>Go back and try another puzzle.</p>\n"
119     "</div> </body> </html>\n")
120
121 # operational part of the program ****
122
123 # get the form parameters
124 form = cgi.FieldStorage()
125
126 if form: # there is a form
127     # initialize sudoku entry string
128     global entries
129     entries = ""
130
131 # process form parameters into the sudoku entry string
132 for i in range(SIZE * SIZE):
133     key = "c" + str(i+1)
134     if key in form:
135         entries += form[key].value
136     else:
137         entries += " "
138
139 # SIMPL send the entry string to the sudoku engine for processing
140 ret = sendPuzzleToEngine()
141 if ret == DO_9:
142     sendResultsToBrowser()
143 elif ret == BAD_ENTRY:
144     problem("At least one puzzle entry sent to the engine program is bad.")
145 elif ret == SOLUTION.FAILURE:
146     problem("Program unable to solve the puzzle.")
147 elif ret == FORK.ERROR:
148     problem("Program cannot solve the puzzle at this time.")
149 elif ret == TOO_MANY.ERROR:
150     problem("Program is too busy solving other puzzles at this time.")
151 elif ret == TIME.ERROR:
152     problem("Program couldn't solve the puzzle within the allowed time.")
153 elif ret == BAD.TOKEN:
154     problem("Unknown message token sent to the solver program.")
155 elif ret == NAME.LOCATE.FAILURE:
156     problem("Problem name locating the solver program.")
157 elif ret == SEND.FAILURE:
158     problem("Problem sending entries to the solver program.")
159 else:
160     problem("Unknown message token from solver program.")
161 else: # no form available
162     problem("There is a form problem: Check your entries.")

```

Lines 124-162 define the operational part of the program whereby the actual step by step actions take place. Looking up at the predefined functions gives the details of reading out the input data from the browser form, creating the SIMPL message to be sent to the Sudoku puzzle solving engine, reading the reply message from the engine, and filling in the browser form for display.

**NOTE:** Where are the files *engine*, *sudoku-cgi.py*, and *sudoku.html* to be found? In the case of *engine*, anywhere that is convenient for you. In the case of *sudoku-cgi.py*, it should be placed in the directory accessed by Apache for CGI programs. Normally, this will be something like /opt/lampp/cgi-bin in the case of a *lampp* installation or possibly /var/www/cgi-bin. As for *sudoku.html*, the place that Apache will look is in the directory often set by the *DocumentRoot* directive. This can be /opt/lampp/htdocs in the case of a *lampp* installation or /var/www/html perhaps. See Appendix L for more on the required Apache configuration details.

### 18.5.5 Possible Problems

1. SIMPL function calls as used between the CGI script *sudoku-cgi.py* and the Sudoku puzzle engine may have trouble communicating if the fifos that are used to trigger off communications and are located in *FIFO\_PATH* have not got the correct read and write permissions. The fifos are made with permissions set to "666". This is to say that all users should have read and write permissions. However, if the umask on the system is set to something else, some of these permissions may be masked off and the CGI program may not have permissions to write to one of the fifos say, thus invalidating SIMPL communications resulting in errors. These errors will be reported back to the browser. If you don't wish to change the umask, then you may have to run:

```
cd $FIFO_PATH  
chmod a+rwx *
```

after step 2. in 18.5.2.

2. You may get a blank screen sent back to the browser. System security may not allow the Apache web server to access the SIMPL modules through the CGI script. You will find several entries to this effect within the /etc/httpd/logs/error\_log file if this is happening. Note that the error\_log file may be located elsewhere depending on your installation. It will say something to the effect that Apache is unable to locate the SIMPL modules for import.

For example, the Apache web server is running on a Linux box equipped with SELinux which is preventing Apache from opening the required mod-

ules according to the SELinux troubleshoot browser which is available from the main menu under the administration moniker. Apache's error\_log describes the problem as one of not being able to locate the said modules for import. This is misleading to say the least. Adjusting SELinux to allow this is the solution to the problem; this can be done under 'main menu/administration/ SELinux Management'

3. Another possibility is that PYTHONPATH and/or FIFO\_PATH are not set. This can be checked by running "cgi\_test()" after the "import cgi" in the top of the CGI script sudoku-cgi.py. All pertinent values and settings will displayed in the browser.

### 18.5.6 In the Real World

Normally the browser, the server, and quite possibly the Sudoku engine program (for example) would be running on different hosts. We have run them on one host for the sake of simplicity. Moreover, if someone wants to learn about these things it is easier to run all of this on one computer and not have to be concerned with networking issues which can easily obscure the situation.

## 18.6 Summary

In this chapter we have discussed extending our locally run Sudoku client/server by:

1. Adding a forking engine capability.
2. Networking the client GUI with the server engine.
3. Adding a Tcl/Tk GUI.
4. Adding a Java/Swing GUI.
5. Using a web browser to provide a GUI via CGI.

In our original spec we defined other possible word puzzles. Completing this ambitious spec is beyond the scope of this book. Suffice to say that those other puzzle engines would be developed along the same lines as the Sudoku program. What is important is that we've used Sudoku to illustrate SIMPL and the SIMPL approach to problem solving. Despite the fact that our spec was rather *concocted*, hopefully the reader is left with the impression that SIMPL can be used to systematically approach a complicated real world spec.



# Chapter 19

## Conclusion

When we were first exposed to the Send/Receive/Reply messaging we found it to be an immensely powerful way to create software. As we got more involved we found that this messaging approach encouraged us to decompose our problems into manageable modules. As we worked on more diverse projects of real world complexity we found that our repository of seed code grew steadily. We believe that Kevin Kelly got it right when he said that complex systems are best grown from simple systems which already work.<sup>1</sup>

As our SIMPL repository grew we recognized that we had repeating patterns in our solutions. We coined the term softwareIC to encapsulate this observation. Like the hardware IC did for the hardware designer, we believe the softwareIC will lower project risk, cost and time to market for the software designer. We have found that SIMPL is a great tool for creating softwareICs.

We also found (through trial and error) that our testing framework helps lower the risks associated with a complex project. We like the fact that with SIMPL we can build our modules and test them locally, confident that with the use of surrogates they will *simply work* when deployed across a network. When it comes to programming languages, we are pragmatic and we like the fact that the SIMPL toolkit allows us to create SIMPL modules in ‘C’, ‘C++’, Tcl, Python, and Java. In future, perhaps the SIMPL library will be extended to still other other languages and formats.

It is no accident that when searching for acronyms for the nascent project that we settled on SIMPL. The KISS (*Keep It Simple Stupid*) principle is central to our programming philosophy and its influence pervades the SIMPL toolkit. Many useful SIMPL applications can be written using only the core

---

<sup>1</sup>Out of Control: The New Biology of Machines, Social Systems, and the Economic World by Kevin Kelly, Addison-Wesley Pub., 1994, ISBN 0-210-48340-8

five functions in the SIMPL library, viz. *name\_attach*, *name\_locate*, *Send*, *Receive*, *Reply*.

We have found that a project specification is a moving target more often than not. As such, it is almost always best to start with the simplest possible subset of features that target users will find useful and code them first. It is always best to engage the users as early as possible in exercising this code. We do this with the confidence that our tokenized message enabled SIMPL modules will allow us to evolve our feature sets in unison with the user's wishes.

We built the SIMPL toolkit because we found it useful. We released the SIMPL toolkit as open source because we think you'll find it useful as well. We acknowledge the contributions of many other programmers who have donated their efforts into the open source pool. As a result, the barrier to entry for a new Linux programmer is as low as it gets. The SIMPL toolkit truly stands on the contributions of many.

Finally, we wrote this book because it was time. It was time to collect our experience and make it available to a new generation of developers. We hope that you will find it helpful.

# Part V

# Appendices



# Appendix A

## Installation

### A.1 Installing SIMPL Core Code

Before you begin you will need to make a couple of decisions about where your SIMPL source tree will exist and where your SIMPL sandbox will exist. The locations of these two areas are governed by a couple of environment variables that you need to set.

- SIMPL\_HOME - locates the source tree (must end with *simpl*). For example: */home/simpl*
- FIFO\_PATH - locates the SIMPL sandbox. For example: */home/fifo*

The SIMPL\_HOME environment variable is only used by the SIMPL build scripts and Makefiles. The FIFO\_PATH environment variable is used internally by the SIMPL library to execute ***name\_attach*** and ***name\_locate*** calls.

The SIMPL source tree has been carefully designed not to *pollute* your system. All SIMPL related files will exist under one of these two directories above.

You are now ready to go on line and grab the latest version of SIMPL source code from the SIMPL Open Source project website:

*<http://www.icanprogram.com/simpl>*

or the Sourceforge repository for SIMPL at:

*<https://sourceforge.net/projects/simpl>*

The SIMPL project is a dynamic Open Source project and releases are made regularly. It is also quite stable.

It is important that you pick up the latest release to capture all the bug fixes that have been done. When in doubt which version to use, the Sourceforge site always contains the latest release.

Once you have downloaded the latest tarball you need to place it at the subdirectory immediately above the SIMPL\_HOME directory. ie.

```
cd $SIMPL_HOME  
cd ..  
<place your SIMPL tarball here>
```

You can undo your tarball by typing:

```
tar -zxvf whateveryourSIMPLtarballnameis.tar.gz
```

At this point you should have a whole bunch of source code sitting at SIMPL\_HOME.

On a Linux system for example, SIMPL\_HOME and FIFO\_PATH are usually exported during system boot via /etc/profile as follows:

```
export FIFO_PATH=/home/fifo  
export SIMPL_HOME=/home/simpl
```

The actual directory locations can of course be anything you want them to be.

## A.2 Building the SIMPL Core Libraries

After installing all the source on your system the next step is to build all this source code into the SIMPL libraries. To do this you will need to:

```
cd $SIMPL_HOME/scripts  
.buildsimpl
```

This should cause the SIMPL source to completely build and install itself. For errors you can examine the \$SIMPL\_HOME/make.out file that this script produces.

Being a library, it is not immediately obvious that SIMPL is operational. One of the subdirectories under the SIMPL\_HOME tree is called **benchmarks**. Some SIMPL executables were built here. They can be used to *time* the SIMPL message passing on your system. We will be using them here to verify that you have SIMPL installed and the environment variables are correctly defined. The buildsimpl script should have caused two executables to be built. Namely,

- \$SIMPL\_HOME/benchmarks/bin/receiver
- \$SIMPL\_HOME/benchmarks/bin/sender

To run this benchmark test you will need to open two text consoles on your system and point each to the \$SIMPL\_HOME/benchmarks/bin directory.

On console 1, logged onto \$SIMPL\_HOME/benchmarks/bin type:

```
receiver -n BOBR
```

where BOBR - is an arbitrary SIMPL name you have chosen for this receiver.

On console 2, logged onto \$SIMPL\_HOME/benchmarks/bin type:

```
sender -n BOBS -r BOBR -t 100000 -s 1024
```

where

- -n BOBS - is the SIMPL name for the sender.
- -r BOBR - is the SIMPL name used for the receiver.
- -t 100000 - send 100000 messages before displaying timing.
- -s 1024 - makes each message 1024 bytes in length (same size for replies).

When you hit enter on console 2 the sender locates the receiver and then marks the time. It then proceeds to send 100000 1k messages to the receiver, each time blocking for the same size reply.

If you see segment faults in the above this is a clear indication that you do not have your FIFO\_PATH environment variable defined in such a way that they are set for each login.

When this preset number of messages has been exchanged the sender calculates and reports the total elapsed time in msec.

You have asked your system to do a substantial piece of work here. Depending on your processor type and speed this may take several tens of seconds to complete. Once you have the number you can easily compute the number of SIMPL messages per second that your system is capable of exchanging.

At this point you will have successfully installed the core SIMPL libraries.

## A.3 SIMPL Public SoftwareICs Code Repository

We discussed the concept of softwareICs at length in the book. The SIMPL project maintains a public repository of several softwareICs. These can be used as seed code for starting any project.

**Step 1:** To obtain the latest softwareICs tarball you'll need to visit the main SIMPL project website at: <http://www.icanprogram.com/simpl>

**Step 2:** Locate and download the softwareICs tarball and place it one level above the \$SIMPL\_HOME directory.

```
cd $SIMPL_HOME  
cd ..  
<place softwareICs tarball here>
```

**Step 3:** Undo that tarball to install the softwareICs code.

```
tar -zxvf simplics.tar.gz
```

**Step 4:** You can now build the softwareICs code by going to the top level area and running the top level make at that location.

```
cd $SIMPL_HOME/softwareICs  
make clobber  
make install
```

This will create a series of executables at \$SIMPL\_HOME/softwareICs/bin.

**Step 5:** Each softwareIC test subdirectory will contain a README file describing how that code can be executed. While this is of immense value in understanding how each of these pieces work, the real value of this software repository is as seed code for any SIMPL project you tackle.

## A.4 SIMPL Testing Framework

The SIMPL testing framework was discussed at length in Chapter 9.

**Step 1:** On the main SIMPL website you find the SIMPL testing framework tarball. Download and install that as you have done before.

```
cd $SIMPL_HOME  
cd ..  
<place simpltest.tar.gz here>
```

**Step 2:** Undo the tarball to expose the SIMPL framework wrapper scripts.

```
tar -zxvf simpltest.tar.gz
```

This will create an \$SIMPL\_HOME/testing tree and place all of the existing SIMPL test scripts there. It will also populate the \$SIMPL\_HOME/scripts area with the SIMPL testing framework wrapper scripts.

**Step 3:** Setup the framework.

You'll need to create an environment variable called TEST\_HOME which will point at your particular project. For now just point this TEST\_HOME at \$SIMPL\_HOME

```
export TEST_HOME=$SIMPL_HOME
```

If the \$SIMPL\_HOME/scripts is in your PATH you can simply type:

```
seetest i
```

If you see a list of the SIMPL tests, your installation has been successful.

## A.5 Python SIMPL

SIMPL also comes with source for a Python SIMPL library. This source is composed of ‘C’ wrappers which are used to create a Python SIMPL module thereby *extending* Python’s capabilities.

It will be necessary for you to have the following items available on your system in order to run Python SIMPL:

1. The Python interpreter *python*. On a Linux system you may find out whether the interpreter is present by typing *which python* at the command line. If you don’t get something like */usr/bin/python*, then either the interpreter is not within your command path or it is not present. In the former case you will have to export the directory location to your command path and in the latter case you will have to obtain a copy of the interpreter.
2. The Python ‘C’ library and headers. On a Linux system you will likely find the necessary files under */usr/include/python2.5* and */usr/lib/python2.5* or under */usr/share*. The 2.5 suffix is an example of a Python version number. Without these files the Python SIMPL module cannot be made.

If you have gotten this far, then all of the above files are present and available on your system; now you have to obtain the SIMPL software.

**Step 1:** From the main SIMPL website download the Python hooks SIMPL tarball onto your system.

**Step 2:** To undo the tarball by perform the following steps:

```
cd $SIMPL_HOME  
cd .. (yes that's up one level from $SIMPL_HOME)  
<place the simplpython.tar.gz here>  
tar -zvxf simplpython.tar.gz
```

This will create an *\$SIMPL\_HOME/python* tree and place all of the source for the code in there.

**Step 3:** Build the code.

This is where most novice SIMPL users have difficulty. The SIMPL Python code is used to create amongst other things a shared library in the form of a module which renders the SIMPL API as Python functions. In order

to successfully compile this module your compiler must have access to the Python source headers and library as was discussed above in the necessary items list. So if your Linux distribution doesn't have those installed you'll need to do that first before proceeding.

If the \$SIMPL\_HOME/scripts is in your command PATH you can simply type:

```
buildsimpl.python
```

Alternatively you can,

```
cd $SIMPL_HOME/python  
make clobber  
make install
```

This will create the Python module and move it to \$SIMPL\_HOME/modules.

**Step 4:** Run the examples. There are some examples of Python SIMPL code in the \$SIMPL\_HOME/python/test directory as well as in the [http://www.icanprogram.com/simpl/eg\\_installation.html](http://www.icanprogram.com/simpl/eg_installation.html) examples tarball.

## A.6 Tcl SIMPL

The SIMPL toolkit comes with hooks for the Tcl language. These hooks let you create SIMPL modules using the Tcl scripting language.

It will be necessary for you to have the following items available on your system in order to run Tcl SIMPL:

1. The Tcl interpreter *wish*. On a Linux system you may find out whether it is present by typing *which wish* at the command line. If you don't get something like */usr/bin/wish* then either the interpreter is not within your command path or it is not present. In the former case you will have to export the directory location to your command path and in the latter case you will have to obtain a copy of the interpreter.
2. The Tcl 'C' library and headers. On a Linux system you will likely find the necessary files under */usr/include/tcl8.4* and */usr/lib/tcl8.4* or under */usr/share*. The 8.4 suffix is the version number. Without these files the Tcl SIMPL module cannot be made.

If you have gotten this far, then all of the above files are present and available on your system; now you have to obtain the SIMPL software.

**Step 1:** From the main SIMPL website download the Tcl tarball.

**Step 2:** To undo the tarball by perform the following steps:

```
cd $SIMPL_HOME  
cd .. (yes that's up one level from $SIMPL_HOME)  
<place the simpltcl.tar.gz here>  
tar -zxvf simpltcl.tar.gz
```

This will create an \$SIMPL\_HOME/tcl tree and place all of the source for the code in there.

**Step 3:** Build the code.

This is where most novice SIMPL users have difficulty. The SIMPL Tcl code is used to create amongst other things a shared library which exposes the SIMPL API as Tcl commands. In order to successfully compile this shared library your compiler must have access to the Tcl source headers as was described above in the necessary items list. So if your Linux distribution doesn't have those installed you'll need to do that first before proceeding.

If the \$SIMPL\_HOME/scripts is in your command PATH you can simply type:

```
buildsimpl.tcl
```

Alternatively you can,

```
cd $SIMPL_HOME/tcl  
make clobber  
make install
```

This will create the Tcl shared library and move it to \$SIMPL\_HOME/lib.

**Step 4:** Run the examples. There are some examples of Tcl SIMPL code in the \$SIMPL\_HOME/tcl/test directory as well as in the [http://www.icanprogram.com/simpl/eg\\_installation.html](http://www.icanprogram.com/simpl/eg_installation.html) examples tar-ball.

# Appendix B

## Language Packages Summary

Language	Package	Linux/Unix/etc. compatible	Windows compatible
'C'	simpl.so	yes	no
Python	csimpl	yes	no
	tclsimpl	yes	yes
Java	Jsimpl	yes	no
	Ssimpl	yes	yes
Tcl	fctclx	yes	yes
	fcgateway	yes	yes

Table B.1: Summary of SIMPL Language Packages and Compatibility



# Appendix C

## C Library Functions

### C.1 Function Synopsis

The following contain descriptions of the ‘C’ SIMPL library calls offered at the time of printing.

<b>int name_attach(char *processName, void (*exitFunc)())</b>
---

*Description:* The name\_attach() function is required by every process that expects to send/receive messages to/from any other process. It is responsible for setting up all of the SIMPL functionality that may be required. It must precede any other SIMPL library calls. The string processName is the desired SIMPL name of the process. This name must be unique on each host but not necessarily on a network. A process that has successfully called this function is often referred to as being *name attached*. The length of processName is defined as follows:

$$0 < \text{length} \leq \text{MAX\_PROGRAM\_NAME\_LEN}$$

If the name is too large it is truncated to MAX\_PROGRAM\_NAME\_LEN to fit. The exitFunc() is some function that the programmer wants to have run when the process exits or in the case of the occurrence of a SIGTERM or similar signal (ie. one that can be trapped). It returns 0 for success and -1 for failure. Failures may occur due to:

1. The name string processName is too short (zero bytes).

2. The environment variable FIFO\_PATH has not been defined.
3. processName is not unique on that particular host.
4. A receive FIFO could not be made.
5. A reply FIFO could not be made.

*Example:* The calling process wants to attach the SIMPL name "receiver".

```
#include <simpl.h>

void main()
{
int id;

// SIMPL name attach
id = name_attach("receiver");
if (id == -1)
{
printf("name attach error=%s\n", whatsMyError());
exit(-1);
}
}
```

### int name\_detach(void)

*Description:* The name\_detach() function removes SIMPL related components from a program that has previously run a successful name\_attach(). This call is made automatically by the atexit() procedure within SIMPL if the program itself doesn't call it directly; so in some ways it could be considered redundant. However, it is recommended if only for good form. It returns 0 on success or -1 on failure. Failure occurs if the calling process had never been name attached.

*Example:* The following process has successfully name attached SIMPL name "receiver" and either wants to detach the SIMPL functionality or it is terminating.

```
#include <simpl.h>

void main()
{
int id;

// SIMPL name attach
```

```
id = name_attach(" receiver");
/*
   body of program
*/
// SIMPL name detach
name_detach();
}
```

### **int child\_detach(void)**

*Description:* The child\_detach() function allows the child process of a forked parent to remove the common SIMPL components that exist in the parent and child processes after forking. This occurs if the parent has been name attached prior to the fork() call. The child\_detach() call should be made prior to the child process running its own name\_attach() if called. It returns 0 for success and -1 for failure. Failure occurs if the parent process was not name attached in the first place.

*Example:* The following name attached parent process has successfully forked a child process.

```
#include <simpl.h>

// parent process
void main()
{
    int id;
    pid_t pid;

    // SIMPL name attach
    id = name_attach("parent");

    // fork child
    pid = fork();
    if (pid < 0) // fork failure
    {
        printf("fork error-%s\n", fn, strerror(errno));
    }
    else if (pid == 0) // child
    {
        myChild();
    }
    else // parent
    {
```

```

/*
    body of program
*/
}
}

// child process
void myChild()
{
int id;

// detach from parent's SIMPL components
child_detach();

// SIMPL name attach the child separately
id = name_attach("child");
/*
    body of program
*/
}

```

### **int sur\_detach(**int** id)**

*Description:* The sur\_detach() function must be called if a process exits after running a successful name\_locate() call for another process on a remote host. The sur\_detach() will cause the surrogates to terminate by sending them SUR\_CLOSE messages. The number of receiver surrogates is stored in a table internal to the sending process. Under normal circumstances this will be called internally by the name\_detach() function in the case of process termination. The id passed into the function is the surrogate receiver FIFO id. This function is called as part of the atexit() function for any process that performs a non-local name\_locate(). It returns 0 for success and -1 for failure. Failures may occur due to:

1. The id passed in is non-sequitor.
2. Communication errors with the receiver surrogate.

*Example:* The process name locates another process called "receiver" on remote host 192.168.1.42 via the RS-232 serial protocol. Later, it breaks the connection made by the name locate call.

```
#include <simpl.h>

void main()
{
int id;

// try to make a remote SIMPL connection
id = name_locate("SIMPL_RS232:192.168.1.42:receiver");
/*
    processing ...
*/
// remote connection no longer needed
sur_detach(id);
/*
    processing ...
*/
}
```

```
int name_locate(char *protocolName:hostName:processName)
```

*Description:* The name\_locate() function must be called prior to a Send() or Trigger() call so that the sender may connect to the receiver it wishes to send to. The protocolName determines the ensuing remote protocol method of data exchange. If this is not set then a default is chosen. The default will be the first available protocol found in the protocol router's table. The hostName is the name of the host that the sender wishes to send to. It may be either a real host name (canonical or alias) or non-existent in the case of a local send. It can also be in the dotted network format. If the local host name/IP is used then SIMPL will run locally. The processName is the name of the receiving process that is name attached on that particular host. It returns the id of the receiver process that a message is to be sent to  $\geq 0$  or -1 for failure. Failures may occur due to:

1. The calling process was never name attached.
2. The protocol name is not supported.
3. The process name is NULL.
4. The HOSTNAME environment variable is not set.
5. Too many colons in the argument (maximum of 2).

And for remote name locates only:

6. Failure to contact the protocolRouter program.
7. No surrogates available.
8. Failure to contact remote surrogate.
9. Failure to find remote receiving program.

*Example One:* This is an example of where the name of the process of interest is running on the local host.

```
#include <simpl.h>

void main()
{
int id;

// no host name implies local host ==> local connection
id = name_locate("localProcessName");
/*
    processing ...
*/
}
```

*OR*

```
#include <simpl.h>

void main()
{
int id;

// host name is the local host ==> local connection
id = name_locate("localHostName:localProcessName");
/*
    processing ...
*/
}
```

*Example Two:* This is an example of where the process of interest is running on a remote host.

```
#include <simpl.h>

void main()
```

```
{
int id;

// default protocol, remote host name ==> remote connection
id = name_locate("remoteHostName:remoteProcessName");
/*
    processing ...
*/
}
```

*OR*

```
#include <simpl.h>

void main()
{
int id;

// specified protocol, remote host name ==> remote connection
id = name_locate("protocolName:remoteHostName:remoteProcessName");
/*
    processing ...
*/
}
```

*Example Three:* This process specifies the protocol type as SIMPL\_TCP, the IP address 192.168.1.101 of the host and the receiver's SIMPL name is receiver1. If the IP address is the same as that of the local host then the communication will be local SIMPL and the protocol field will be ignored.

```
#include <simpl.h>

void main()
{
int id;

// specified protocol and dotted network name ==> remote connection
id = name_locate("SIMPL_TCP:192.168.1.101:receiver1");
/*
    processing ...
*/
}
```

---

```
int Receive(char **ptr, void *inArea, unsigned maxBytes)
```

---

*Description:* The Receive() function receives messages from senders in a memory area pointed to by inArea and no larger than that specified by maxBytes. In the case that inArea is NULL the maxBytes is ignored and the actual copying of the message can then be done via the simplRcopy() call. In this way an undetermined amount of memory can be accounted for by using dynamic allocation as opposed to some global or stack buffer if desired. The \*ptr is set in the Receive() and uniquely identifies the sender. This value is later used for replies. This function returns a value of  $\geq 0$  for the size of an incoming message or  $\leq -2$  which indicates a proxy has been received or -1 for a failure. In the case of a proxy being sent you must call the function returnProxy(int ret) to obtain the true value of the proxy where ret is the Receive() return value. Failures can arise from:

1. The calling process was never name attached.
2. FIFO trigger problems.
3. Shared memory (shmem) problems.

*Example One:* This process is receiving a message with a size in bytes that is well known.

```
#include <simpl.h>

void main()
{
int msgSize;
char *sender;
char inArea[1024];
int maxBytes = 1024;

// SIMPL name attach
name_attach("receiver");
// receive messages
msgSize = Receive(&sender, inArea, maxBytes);
/*
    process message ...
*/
}
```

*Example Two:* This process is receiving a message with a size in bytes that is not well known.

```
#include <simpl.h>

void main()
{
int msgSize;
char *sender;
char *inArea;

// SIMPL name attach
name_attach("receiver");
// receive messages
msgSize = Receive(&sender, NULL, 0);
// dynamically allocate sufficient memory
inArea = malloc(msgSize);
// copy the message into the allocated memory
simplRcopy(sender, inArea, msgSize);
/*
    process message ...
*/
}
```

*Example Three:* This process is receiving a proxy.

```
#include <simpl.h>

void main()
{
int proxy;
int msgSize;
char *sender;
char inArea[1024];
int maxBytes = 1024;

// SIMPL name attach
name_attach("receiver");
// receive messages
msgSize = Receive(&sender, inArea, maxBytes);
if (msgSize < -1)
{
    proxy = returnProxy(msgSize);
}
/*
    react to proxy ...
*/
// a Reply() is unnecessary for a proxy
}
```

```
int Send(int id, void *out, void *in, unsigned outSize, unsigned inSize)
```

*Description:* The Send() function is a blocked send. This means that the calling program expects a reply from the receiver and waits until it gets one. The id is the result of a prior call to name\_locate() with regard to the receiver of interest. The function sends a message to the receiver pointed to by out with message size = outSize in bytes. It expects a reply message from the receiver to be placed in memory pointed to by in with a size no larger than inSize in bytes. The function returns the size of the reply message  $\geq 0$  in bytes for success and -1 for a failure. Failures can occur due to:

1. The calling process was never name attached.
2. name\_locate() was not run prior to the Send call.
3. FIFO problems.
4. Reply errors.
5. Reply message larger than inSize.

*Example One:* This process is sending to another local process called 'receiver' and is not expecting any return message.

```
#include <simpl.h>

void main()
{
    int receiverId;
    char outArea[1024];
    int outBytes = 1024;

    // SIMPL name attach
    name_attach("sender");
    // make connection
    receiverId = name_locate("receiver");
    // send message to receiver expecting nothing in reply
    Send(receiverId, outArea, NULL, outBytes, 0);
    /*
        await null reply message from receiver ...
    */
}
```

*Example Two:* This process is sending to another local process called 'receiver' and is expecting any return message.

```
#include <simpl.h>

void main()
{
int receiverId;
char outArea[1024];
char inArea[1024];
int inBytes = 100;
int outBytes = 512;

// SIMPL name attach
name_attach("sender");
// make connection
receiverId = name_locate("receiver");
// send a mesage to receiver expecting a reply no larger than inBytes
Send(receiverId, outArea, inArea, outBytes, inBytes);
/*
   await and process reply message from receiver ...
*/
}
```

**int Reply(char \*ptr, void \*outArea, unsigned size)**

*Description:* The Reply() function responds a return message from a receiver to a blocked sender indicated by ptr. The reply message is pointed to by outArea and is of size in bytes. The function returns 0 for success and -1 for failure. Failures can occur due to:

1. The calling process was never name attached.
2. FIFO errors.

*Example One:* This process has received a message from 'sender' and is replying a null response because the sender is not expecting any return reply message.

```
#include <simpl.h>

void main()
{
int msgSize;
char *sender;
char inArea[1024];
```

```

int maxBytes = 1024;

// SIMPL name attach
name_attach("receiver");
// receive messages
msgSize = Receive(&sender, inArea, maxBytes);
// reply null message
Reply(sender, NULL, 0);
/*
   process message ...
*/
}

```

*Example Two:* This process has received a message from 'sender' and is replying some sort of response.

```

#include <simpl.h>

void main()
{
int msgSize;
char *sender;
char inArea[1024];
char *outArea[1024];
int maxBytes = 1024;

// SIMPL name attach
name_attach("receiver");
// receive messages
msgSize = Receive(&sender, inArea, maxBytes);
/*
   process message and construct reply message...
*/
// reply message back to sender
Reply(sender, outArea, 1024);
}

```

<b>int Relay(char *sender, int channelID)</b>
---

*Description:* The Relay() function forwards a message while keeping the original sender blocked. The original message is pointed to by sender. The channelID is the SIMPL ID returned from a name\_locate call for the intended destination of the relay operation. The function returns 1 for success and -1 for failure. Failures can occur due to inability to write to the channelID.

*Example One:* This process has received a message from 'sender' and is relaying that same message to second process.

```
#include <simpl.h>

void main()
{
int msgSize;
char *sender;
int channelID;
char inArea[1024];
int maxBytes = 1024;

// SIMPL name attach
name_attach("receiver");
// open channel to second receiver process
channelID=name_locate("receiver2");
// receive messages
msgSize = Receive(&sender ,inArea , maxBytes );
/*
process the message
*/
// relay to second receiver
Relay(sender , channelID );
}
```

*Example Two:* This process has received a message from 'sender' and is leaving the message in shared memory and then relaying it to second receiver.

```
#include <simpl.h>

void main()
{
int msgSize;
char *sender;
int channelID;
char *sArea;
char *outArea[1024];
int maxBytes = 1024;

// SIMPL name attach
name_attach("receiver");
// open channel to second receiver process
channelID=name_locate("receiver2");
// receive messages
msgSize = Receive(&sender ,NULL,0);
// line up pointer to message in shared memory
sArea=whatsThisShmPtr(sender );
```

```

/*
  process message at sArea...
*/
// relay message to second receiver
Relay(sender , channelID );
}

```

### **int Trigger(int id, int proxy)**

*Description:* The Trigger() function sends a proxy identified as an int to a receiving process identified by a prior call to name\_locate() giving id. It is important that the proxy *MUST* be an integer > 0 AND < 7FFFFFFF. This can be thought of as a kick to the receiver and requires no reply. This function returns 0 on success and -1 for a failure. Failures can occur due to:

1. The calling process was never name attached.
2. Receiver does not exist.
3. FIFO errors.
4. Proxy number is out of range.

*Example:* This process is triggering another local process called 'receiver'.

```

#include <simpl.h>

void main()
{
int proxy = 10;
int receiverId;

// SIMPL name attach
name_attach("sender");
// make connection to receiver
receiverId = name_locate("receiver");
// send a proxy to receiver
Trigger(receiverId , proxy );
}

```

```
char *whatsMyName(void)
```

*Description:* The whatsMyName() function gets the SIMPL name of the calling process. It returns the SIMPL name on success and NULL on failure. Failure can occur if the calling process was never name attached.

*Example:* This process name attached itself as 'sender'. The result of the printf should be 'sender'.

```
#include <simpl.h>

void main()
{
    // SIMPL name attach
    name_attach(" sender");

    // print my SIMPL name
    printf("My SIMPL name is %s\n", whatsMyName());
}
```

```
int whatsMyRecvfd(void)
```

*Description:* The whatsMyRecvfd() function gets the receive FIFO file descriptor of the calling process. If one has not been assigned, one will be made. The function returns the receive FIFO file descriptor on success and -1 on failure. Failures can occur due to:

1. The calling process was never name attached.
2. The FIFO could not be opened for read/write.

*Example:* This process name attached itself as 'sender'. The result of the whatsMyRecvfd() should be a file descriptor to its SIMPL receive FIFO. In this example we are allowing the process to 'kick' on either an incoming SIMPL message or a timer.

```
#include <simpl.h>

void main()
{
    int ret;
    int fd;
    fd_set watchset;
    fd_set inset;
```

```
struct timeval tv;
struct timeval *timeoutPtr;

// SIMPL name attach
name_attach("sender");

// get receive FIFO file descriptor
fd = whatsMyRevcfd();

// set select parameters
FD_ZERO(&watchset);
FD_SET(fds[0], &watchset);

// set timer parameters
tv.tv_sec = 10;
tv.tv_usec = 0;
timeoutPtr = &tv;

// let select kick on the file descriptor or the timer
ret = select(fd + 1, &watchset, NULL, NULL, timeoutPtr);
/*
     react on select ...
*/
}
```

### int whatsMyReplyfd(void)

*Description:* The whatsMyReplyfd() function finds the reply FIFO file descriptor of the calling process. If one has not been assigned, one will be made. The function returns the reply FIFO file descriptor on success and -1 on failure. Failures can occur due to:

1. The calling process was never name attached.
2. The FIFO could not be opened for read/write.

*Example:* See the example in whatsMyRevcfd() and simply substitute the whatsMyReplyfd() call for the whatsMyRevcfd() call.

**char \*whatsThisShmPtr(char \*)**

*Description:* The whatsThisShmPtr() function points to the position in shared memory where the SIMPL message begins. This is typically used following a Receive(&sender,NULL,0) call.

*Example:* See the example 2 for the Relay().

**char \*whatsMyShmPtr(void)**

*Description:* The whatsMyShmPtr() function points to SIMPL message in a SIMPL sender's shared memory. This typically follows a Send(ID, outbuf, NULL, sbytes, rbytes) call or precedes a Send(ID, NULL, inbuff, sbytes, rbytes) call.

*Example:* This process name attached itself as 'sender'. The result of the whatsMyShmPtr() should be a pointer to the SIMPL message area in the sender's shared memory where the receiver's response will also live.

```
#include <simpl.h>

void main()
{
int ret;
int channelID;
char outArea[1024];
char *rArea;
int outBytes;
int inBytes=512;

// SIMPL name attach
name_attach("sender");

// locate the receiver
channelID=name_locate("receiver");

/*
build up the message in outArea
and set outBytes to actual message size
*/
// send a mesage to receiver expecting a reply no larger than inBytes
Send(receiverId, outArea, NULL, outBytes, inBytes);
rArea=whatsMyShmPtr();
```

```
/*
process reply message from receiver at rArea ...
```

**NOTE:**

*Care must be taken to not insert any other function which does a Send() (eg. trace logger) until the rArea is processed, because those sends will overwrite the shared memory area.*

```
*/  
}
```

**char \*whatsMyError(void)**

*Description:* The whatsMyError() function returns a descriptive string based on the last value of \_simpl\_errno. This is analogous to the well known errno/strerror(errno) combination in ‘C’. The function returns a pointer to a the error string.

*Example:* See the example in name\_attach().

**void simplRcopy(char \*src, void \*dst, unsigned size)**

*Description:* The simplRcopy() function copies size bytes from the memory pointed to by src to dst. It works in conjunction with a Receive(&id, NULL, 0) type call in order to extract the message data after the size of the message is known. The function returns nothing.

*Example:* See Example Two in Receive().

**void simplScopy(void \*dst, unsigned size)**

*Description:* The simplScopy() copies size bytes from the memory pointed to by a sender’s shmem pointer to dst. It works in conjunction with a Send(fd, &buf, NULL, outSize, inSize) call in order to extract the reply message data after the message is known. Note that inSize must be set to some maximal amount or internal checking will reject the transaction. The function returns nothing.

*Example:* This process is sending to another local process called ‘receiver’ but does not know how much memory to allocate for the reply message.

```
#include <simpl.h>

void main()
{
int receiverId;
char outArea[100];
int outSize = 100;
int inSize;
char *inArea;

// SIMPL name attach
name_attach("sender");
// make connection
receiverId = name_locate("receiver");
// send a message to the receiver expecting nothing in reply
inSize = Send(receiverId, outArea, outSize, NULL, 0);
// dynamically allocate adequate memory for the reply message
inArea = malloc(inSize);
// now copy the message
simplScopy(inArea, inSize);
/*
process the reply
*/
}
```

**int simplReplySize(char \*ptr)**

*Description:* The simplReplySize() function reports the maximum size of the reply message expected by a sender. It is called between Receive() and Reply() calls. It returns the size of the expected reply message which is  $\geq 0$ .

*Example:* This process receives a message from "sender" and then checks the maximum size of the expected reply.

```
#include <simpl.h>

void main()
{
int nBytes;
int yBytes;
char *sender;
char inArea[1024];

// receive message from the local sender
```

```

nBytes = Receive(&sender , inArea , 1024);
if (nBytes == -1)
{
    printf(" receive error-%s\n" , whatsMyError());
    exit(-1);
}

// what is the maximum size of the expected reply message?
yBytes = simplReplySize(sender);
/*
    processing ...
*/
}

```

<b>void simplSetSenderParms(char *sender, SIMPL_REC *rec)</b>
---

*Description:* The simplSetSenderParms() function stores the relevant SIMPL information with respect to "sender" in the structure "rec". This information can later be used to check on "sender's" veracity.

*Example:* This process receives a message from "sender" and later checks to see whether "sender" is still a valid SIMPL process.

```

#include <simpl.h>

void main()
{
int nBytes;
SIMPL_REC senderInfo;
char *sender;
char inArea[1024];

// receive message from the local sender
nBytes = Receive(&sender , inArea , 1024);
if (nBytes == -1)
{
    printf(" receive error-%s\n" , whatsMyError());
    exit(-1);
}

// get sender information
simplSetSenderParms(sender , senderInfo);

/*
    processing ...

```

```
/*
// been away doing other things; is "sender" still awaiting a reply?
if (simplCheckProcess(&senderInfo) == -1)
{
    // sender has gone, no need to send a reply
    exit(0);
}
```

**void simplCheckProcess(SIMPL\_REC \*rec)**

*Description:* The simplCheckProcess() function checks to see whether a process is still alive. It returns a 0 if the process is alive and a -1 if not.

*Example:* See the example in simplSetSenderParms().

**int returnProxy(int proxyNumber)**

*Description:* The returnProxy() function returns the true value of a proxy just received by a Receive() call. proxyNumber is the value  $\leq -2$  returned by the instance of the Receive() call. The proxy number returned will be  $\geq 1$ . This curious arrangement arises due to the fact that the return value of the Receive() function is  $\geq 0$  for messages, null or otherwise. Failures are indicated by -1. This leaves only values  $\leq -1$  if we want to keep Receive() as a simple integer function. Accordingly, when a sender uses Trigger() to send a proxy with a value of 7 say (recall that proxies must be an integer 0), the Trigger() function sends it to the receiver as -7. The Receive() function will then return a value of -8. Plugging this value into returnProxy() returns 7, the value of the original proxy.

*Example:* This example shows the mechanics of a proxy sender and a proxy receiver. The sender sends a proxy equal to 1.

```
#include <simpl.h>

void sender()
{
    int proxy = 1;
    int receiverId;

    // SIMPL name attach
```

```
name_attach("sender");
// name locate receiver process
receiverId = name_locate("receiver");
// send the receiver the proxy
Trigger(receiverId, proxy);
}

void receiver()
{
int nBytes;
int proxy;
char inArea[1024];
char *sender;

// SIMPL name attach
name_attach("receiver");

// receive messages from senders
nBytes = Receive(&sender, inArea, 1024);
if (nBytes == -1)
{
    printf("receive error-%s\n", whatsMyError());
    exit(-1);
}
// is this a proxy?
if (nBytes < -1)
{
    // get the value of the proxy
proxy = returnProxy(nBytes);
printf("proxy=%d\n", proxy);
// printf will display "proxy=1"
}
}
```

## C.2 Function Summary Table

A table of all of the currently available ‘C’ library SIMPL functions follows for handy reference. **NOTE:** The value S\_REC (or SIMPL\_REC) in the table is a structure containing various parameters (such as the SIMPL name) pertaining to a SIMPL process and is normally used internally by the SIMPL library.

Function	Purpose	Return
int name_attach(char *, void *)	Start SIMPL functionality	0/-1
int name_detach(void)	End SIMPL functionality	0/-1
int child_detach(void)	End parent/child SIMPL connect	0/-1
int sur_detach(int)	Release a surrogate from service	0/-1
int name_locate(char *)	Make a SIMPL connection	$\geq 0/-1$
int Receive(char *, void *, uint)	Receive a SIMPL message	int $\geq 0$
int Send(int, void *, void *, uint, uint)	Send a SIMPL message	int $\geq -1$
int Reply(char *, void *, uint)	Reply a SIMPL message	0/-1
int Relay(char *, int)	Forward a SIMPL message	0/-1
int Trigger(int, int)	Send a proxy	0/-1
char *whatsMyNameTrigger(void)	Return SIMPL name	string
int whatsMyRecvfd(void)	Return receive fifo fd	int $\geq 0$
int whatsMyReplyfd(void)	Return reply fifo fd	int $\geq 0$
char *whatsMyRecvPtr(char *)	Return start of receive message	pointer
char *whatsMyReplyPtr(void)	Return start of reply message	pointer
char *whatsMyError(void)	Return SIMPL error	string
void simplRcopy(char *, void *, uint)	Inter-memory copy	nil
void simplScopy(void *, uint)	Inter-memory copy	nil
int simplReplySize(char *)	Return max reply message size	int $\geq 0$
void simplSetSenderParms(char *, S_REC *)	Store sender Simpl info	nil
void simplCheckProcess(S_REC *)	Checks process’ state	nil
int returnProxy(int)	Return the value of a proxy	int $\geq 1$

Table C.1: Summary of ‘C’ Functions



# Appendix D

## Arduino Library Functions

### D.1 Function Synopsis

The following contain descriptions of the Arduino SIMPL library calls offered at the time of printing.

```
int nameAttach(char *sn, unsigned port, EthernetClient *client, char *pn)
```

*Description:* The `nameAttach()` function is required by every process that expects to send/receive messages to/from any other process. It is responsible for setting up all of the SIMPL functionality that may be required. It must precede any other SIMPL library calls. The string *sn* is the name of the host where the tclSurrogate program is running and is often in the form of a URL. Alternatively, it may also take the form of a dotted IP. For example, *sn* may be something like *www.icanprogram.ca* or *192.168.1.1*. The *port* is the port number of the host that the tclSurrogate is listening on. The *client* is an `EthernetClient` class object. The string *pn* is the SIMPL name by which this program wishes to be known by. SIMPL names must be unique on each host, in this case the host is an Arduino so there will be only one program running. In the case of the Arduino, this function is normally run from the usual `setup()` function.

The length of *pn* is defined as follows:

$$0 < \text{length} \leq \text{MAX\_PROGRAM\_NAME\_LEN}$$

It returns 0 for success and -1 for failure. Failures may occur due to:

1. The name string *pn* is too short (zero bytes).

2. The name string pn is too long.
3. Various socket related errors (DNS, connection etc.).
4. Various tclSurrogate problems.

*Example 1:* The calling program wants to attach the SIMPL name "ARDUINO".

```
#include <SPI.h>
#include <Ethernet.h>
#include <Simpl.h>

EthernetClient client;

void setup()
{
const unsigned port = 8000; // tclSurrogate is babysitting this port
char *serverName = "www.icanprogram.ca"; // where tclSurrogate lives
byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77}; // MAC address

// start serial messaging
Serial.begin(9600);
Serial.println();
Serial.println("starting up");

// start ethernet
if (!Ethernet.begin(mac))
{
  Serial.println("error on ethernet");
  while(1);
}
// give the ethernet shield a chance to initialize
delay(1000);

// remotely SIMPL name attach
if (nameAttach(serverName, port, &client, "ARDUINO") == -1)
{
  Serial.println(whatsMyError());
  client.stop();
  while(1);
}
```

*Example 2:* The same as the previous example with the exception that the host where the tclSurrogate lives is located by an IP address rather than a URL.

```
#include <SPI.h>
#include <Ethernet.h>
#include <Simpl.h>

EthernetClient client;

void setup()
{
    const unsigned port = 8000; // tclSurrogate is babysitting this port
    char *server = "192.168.1.10"; // where tclSurrogate lives
    byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77}; // MAC address

    // start serial messaging
    Serial.begin(9600);
    Serial.println();
    Serial.println("starting up");

    // start ethernet
    if (!Ethernet.begin(mac))
    {
        Serial.println("error on ethernet");
        while(1);
    }
    // give the ethernet shield a chance to initialize
    delay(1000);

    // remotely SIMPL name attach
    if (nameAttach(server, port, &client, "ARDUINO") == -1)
    {
        Serial.println(whatsMyError());
        client.stop();
        while(1);
    }
}
```

**int nameDetach(void)**

*Description:* The nameDetach() function removes SIMPL related components from a program that has previously run a successful nameAttach(). It returns 0 on success or -1 on failure. Failure occurs if the calling process had never been name attached or is no longer connected to the tclSurrogate.

*Example:* The following process has successfully name attached SIMPL name ARDUINO and needs to detach the SIMPL functionality on failure of its sendSignal() function.

```
#include <Ethernet.h>
#include <Simpl.h>

EthernetClient client;

void setup()
{
const unsigned port = 8000; // tclSurrogate is babysitting this port
char *serverName = "www.icanprogram.ca"; // where tclSurrogate lives
byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77}; // MAC address

// start ethernet
if (!Ethernet.begin(mac))
{
  while(1);
}
delay(1000);

// remotely SIMPL name attach
if (nameAttach(serverName, port, &client, "ARDUINO") == -1)
{
  client.stop();
  while(1);
}

// signal the world
if (sendSignal() == -1)
{
  // problem - name detach SIMPL, disconnect from server, spin
  nameDetach()
  client.stop();
  while(1);
}
```

<b>int nameLocate(EthernetClient *client, char *pn)</b>
---

*Description:* The nameLocate() function must be called prior to a Send() call so that the sender may connect to the receiver it wishes to send to. The *client* is the Ethernet class object used for connecting the Arduino to a network and/or the web via an Ethernet shield. The string *pn* is the name of the receiving process

that is name attached via the tclSurrogate. Since this function must be run prior to any messaging, it is usually found in the setup() function area of an Arduino program. It returns the result of trying to connect to the receiver program; 0 for success and -1 for failure. Failures may occur due to:

1. The calling program was never name attached.
2. The program name is NULL.
3. The program name is too long.
4. Problems with tclSurrogate communications.
5. Failure to find remote receiving program.

*Example:* This is an example of where the Arduino program is called "SENDER" and is looking to send messages to a program called "RECEIVER".

```
#include <Ethernet.h>
#include <Simpl.h>

EthernetClient client;
long receiverId;

void setup()
{
const unsigned port = 8000;
char *serverName = "www.icanprogram.ca";
byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77};

// start ethernet
if (!Ethernet.begin(mac))
{
  while (1);
}
delay(1000);

// remotely SIMPL name attach to the designated server and port
if (nameAttach(serverName, port, &client, "SENDER") == -1)
{
  client.stop();
  while (1);
}

// name locate the receiver process desired for messaging
receiverId = nameLocate(&client, "RECEIVER");
if (receiverId == -1)
```

```
{
nameDetach(&client );
client.stop();
while(1);
}
```

<b>int Receive(EthernetClient *client, long *senderId, void *iMsg, unsigned iSize)</b>
--

*Description:* The Receive() function receives messages from senders in a memory area pointed to by *iMsg*. The maximum message size is given by *iSize* in bytes and is normally the size of the memory buffer. The *client* is the Ethernet class object used for connecting the Arduino to a network and/or the web via an Ethernet shield. The *senderId* is set in the Receive() and uniquely identifies the sender. This value is later used for replies. This function returns a value of  $\geq 0$  for the size of an incoming message or -1 for a failure. Failures can arise from:

1. The calling process was never name attached.
2. Garbled message.
3. Message too large for available memory.

*Example:* This process is receiving a floating point number and replying that number multiplied by two.

```
#include <SPI.h>
#include <Ethernet.h>
#include <Simpl.h>

// structure definition of received/replied message
typedef struct
{
    int num;
} MSG;

EthernetClient client;

void setup()
{
const unsigned port = 8000; // port for tclSurrogate
char *serverName = "www.icanprogram.ca"; // where tclSurrogate lives
byte mac[] = {0x90, 0xA2, 0xDA, 0xD, 0xD, 0x77}; // MAC address
```

```
// start serial messaging
Serial.begin(9600);
Serial.println();
Serial.println("starting up");

// start ethernet
if (!Ethernet.begin(mac))
{
    Serial.println("error on ethernet");
    while(1);
}
// give the ethernet shield a chance to initialize
delay(1000);

// remotely SIMPL name attach the name "ARDUINO"
if (nameAttach(serverName, port, &client, "ARDUINO") == -1)
{
    Serial.println(whatsMyError());
    client.stop();
    while(1);
}

Serial.println("name attach OK");
Serial.println("initialization complete");
}

void loop()
{
long senderId; // the unique id of the sending program
MSG msg; // incoming/outgoing message variable
int ret; // function return value

// check for an incoming message
ret = Receive(&client, &senderId, &msg, sizeof(MSG));
if (ret == -1) // problem
{
    Serial.println(whatsMyError());
    nameDetach(&client);
    while(1);
}
else if (ret > 0) // incoming message
{
    // reply to the remote sender
    msg.num *= 2;
    if (Reply(&client, &senderId, &msg, sizeof(MSG)) == -1)
    {
```

```

        Serial.println(whatsMyError());
        nameDetach(&client);
        while(1);
    }
}
else
{
    // 0 returned ==> nothing pending so carry on looping
}
}

```

```
int Send(EthernetClient *cli, long *id, void *oM, void *iM, unsigned oS, unsigned iS)
```

*Description:* The Send() function is a blocked send. This means that the calling program expects a reply from the receiver and waits until it gets one. The *cli* is the Ethernet class object used for connecting the Arduino to a network and/or the web via an Ethernet shield. The *id* is the result of a prior call to nameLocate() with regard to the receiver of interest. The function sends a message to the receiver pointed to by *oM* with message size = *oS* in bytes. It expects a reply message from the receiver to be placed in memory pointed to by *iM* with a size no larger than *iS* in bytes. The function returns the size of the reply message  $\geq 0$  in bytes for success and -1 for a failure. Failures can occur due to:

1. The calling process was never name attached.
2. nameLocate() was not run prior to the Send call so *id* is not valid.
3. The receiver has gone away.
4. Communication errors.
5. Reply message larger than inSize.

*Example One:* This program called "SENDER" sends one hundred numbers (one at a time) to the receiver called "NUMRECEIVER" and expects a number in return which is double the original.

```

#include <Ethernet.h>
#include <Simpl.h>

// structure definition of the received/replied message

```

```
typedef struct
{
    float state;
} MSG;

EthernetClient client;
long rId;

void setup()
{
    const unsigned port = 8000;
    char *serverName = "www.icanprogram.ca";
    byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77};

    // start ethernet
    if (!Ethernet.begin(mac))
    {
        Serial.println("error on ethernet");
        while (1);
    }
    // give the ethernet shield hardware an opportunity to initialize
    delay(1000);

    // remotely SIMPL name attach to the designated server and port
    if (nameAttach(serverName, port, &client, "SENDER") == -1)
    {
        Serial.println(whatsMyError());
        client.stop();
        while (1);
    }

    // SIMPL name locate the receiver process desired for messaging
    rId = nameLocate(&client, "NUMRECEIVER");
    if (rId == -1)
    {
        Serial.println(whatsMyError());
        nameDetach(&client);
        client.stop();
        while (1);
    }

    Serial.println("Initialization is complete.");
}

void loop()
{
```

```

int ret;
MSG msg;

// send numbers 1–100 and receive their doubled value
for (int i = 1; i <= 100; i++)
{
    msg.num = (float)i;
    ret = Send(&client, &rId, &msg, &msg, sizeof(MSG), sizeof(MSG));
    if (ret == -1)
    {
        Serial.println(whatsMyError());
        break;
    }

    Serial.print("sent ");
    Serial.println(i);
    Serial.print("received ");
    Serial.println(msg.num);
}

// disconnect from the remote host
nameDetach(&client);
Serial.println("program finished ... going to spin now");
while(1);
}

```

*Example Two:* This process called "SENDER" is sending a device 'state' read by getState() to a receiver called "STATERECEIVER". It expects a NULL reply in return.

```

#include <Ethernet.h>
#include <Simpl.h>

// structure definition of the message
typedef struct
{
    long num;
} MSG;

EthernetClient client;
long rId;

void setup()
{
const unsigned port = 8000;
char *serverName = "www.icanprogram.ca";
byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77};

```

```
// start ethernet
if (!Ethernet.begin(mac))
{
    while (1);
}

// give the ethernet shield hardware an opportunity to initialize
delay(1000);

// remotely SIMPL name attach to the designated server and port
if (nameAttach(serverName, port, &client, "SENDER") == -1)
{
    client.stop();
    while (1);
}

// SIMPL name locate the receiver process desired for messaging
rId = nameLocate(&client, "STATERECEIVER");
if (rId == -1)
{
    nameDetach(&client);
    client.stop();
    while (1);
}
}

void loop()
{
    int ret;
    MSG msg;

    // send state to the state receiver program
    while (1)
    {
        // get the state of a device
        msg.state = getState();

        // send the state to the STATERECEIVER
        ret = Send(&client, &rId, &msg, NULL, sizeof(MSG), 0);
        if (ret == -1)
        {
            // error on send
            break;
        }
    }
}
```

```
// disconnect from the remote host  
nameDetach(&client);  
}
```

```
int Reply(EthernetClient *client, long *id, void *oMsg, unsigned oSize)
```

*Description:* The Reply() function responds a return message from a receiver to a blocked sender, thus unblocking the sender. The reply message is pointed to by *oMsg* and is of *oSize* in bytes. The function returns 0 for success and -1 for failure. Failures can occur due to:

1. The calling process was never name attached.
2. Communication errors.

*Example:* This program turns on/off each of a bank of eight LEDs according to a message received from a remote sender program. The states of the LEDs are controlled by the incoming message. The reply message is referred to as a NULL reply because it has no content. The sender is not expecting any sort of information from the return message so, the NULL message merely releases the send block and assures the sender that its message was received. Note the Reply() in the example used in Receive() above for a nonzero replied message.

```
#include <SPI.h>  
#include <Ethernet.h>  
#include <Simpl.h>  
  
// message token definition  
#define BRINKY_RITE 0  
  
// structure definition of the received message  
typedef struct  
{  
    unsigned short token; // message token indicating what to do  
    unsigned char state; // state of the LEDs; on/off  
} MSG;  
  
EthernetClient client;  
  
void setup()  
{
```

```
const unsigned port = 8000; // port the tclSurrogate is waiting on
char *serverName = "www.icanprogram.ca"; // host of the tclSurrogate
byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77}; // MAC address

// start serial messaging
Serial.begin(9600);
Serial.println();
Serial.println("starting up");

// start ethernet
if (!Ethernet.begin(mac))
{
    Serial.println("error on ethernet");
    while(1);
}
delay(1000);

// remotely SIMPL name attach the name "ARDUINO" on raspi
if (nameAttach(serverName, port, &client, "ARDUINO") == -1)
{
    Serial.println(whatsMyError());
    client.stop();
    while(1);
}

// initialize the digital pins as an output
for (int ledPin = 2; ledPin <= 9; ledPin++)
{
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, HIGH);
}
delay(3000);
for (int ledPin = 2; ledPin <= 9; ledPin++)
{
    digitalWrite(ledPin, LOW);
}
Serial.println("initialization complete");
}

void loop()
{
long senderId;           // the unique id of the sending program
MSG msg;                 // incoming/outgoing message variable
int ret;                  // function return value
unsigned char state;     // 8 led state masking variable
```

```
// check for an incoming message
ret = Receive(&client , &senderId , &msg , sizeof(MSG));
if (ret == -1) // problem
{
    Serial.println(whatsMyError());
    nameDetach(&client);
    while(1);
}
else if (ret > 0) // incoming message
{
    // reply instantly to the remote sender
    if (Reply(&client , &senderId , NULL, 0) == -1)
    {
        Serial.println(whatsMyError());
        nameDetach(&client);
        while(1);
    }

    // check the message token for veracity
    if (msg.token != BRINKY_RITE)
    {
        Serial.println("Incorrect msg token");
        nameDetach(&client);
        while(1);
    }

    // set the blinky lights according to the message state variable
    for (int ledPin=2, state=1; ledPin <= 9; ledPin++, state <<= 1)
    {
        digitalWrite(ledPin , msg.state & state);
    }
}
```

**char \***whatsMyError(**void**)

*Description:* The whatsMyError() function returns a descriptive string based on the sort of error encountered. Note its use in the preceding examples.

## D.2 Function Summary Table

A table of all of the currently available ‘C’ library SIMPL functions for the Arduino follows for handy reference.

Function	Purpose	Return
int nameAttach(char *, unsigned, EthernetClient *, char *)	Start SIMPL	0/-1
int nameDetach(EthernetClient *)	End SIMPL	0/-1
int nameLocate(EthernetClient *, char *)	Make a connection	$\geq 0/-1$
int Send(EthernetClient *, long *, void *, void *, unsigned, unsigned)	Send a message	int $\geq -1$
int Receive(EthernetClient *, long *, void *, unsigned)	Receive a message	int $\geq 0$
int Reply(EthernetClient *, long *, void *, unsigned)	Reply a message	0/-1
char *whatsMyError(void)	Return error	string

Table D.1: Summary of ‘C’ Functions for Arduino



# Appendix E

## Python Module Functionality

### E.1 Libraries and Modules

There are two Python programming modules that offer SIMPL functionality. The reason for this variety is that each module was designed to address a specific situation and/or programming style. We will discuss each module in turn.

The SIMPL functionality may called directly as functions or as an instance of the SIMPL class and its accompanying methods..

#### E.1.1 `csimpl/simpl2.so/simpl3.so`

The simpl2.so and the simpl3.so libraries are Python/‘C’ extension libraries. One or the other is called when a Python script imports the *csimpl* module. That is, for Python versions 2.X and 3.X, simpl2.so and simpl3.so will be imported respectively depending on the version of the Python interpreter being used.

Hence, the csimpl module contains SIMPL functions/methods that are based on ‘C’ extensions. That is, they are built in conjunction with and rely upon the SIMPL ‘C’ library. The Python function names are not necessarily the same as their ‘C’ counterparts due to naming style differences between ‘C’ and Python.

The functionality may be divided into three groups:

1. SIMPL functions for interprocess communications.
2. packing/unpacking functions for constructing/deconstructing messages.
3. auxiliary functionality.

This varied functionality is described below.

The `csimpl` module was written with Unix, Linux, Solaris, MAC OS X etc. platforms in mind. The name *csimpl* comes from ‘C’ extension *simpl*.

### E.1.2 `tclsimpl`

The `tclsimpl` module is completely written in Python and allows SIMPL modules to communicate with each other via the tclSurrogate TCP/IP protocol. (See Appendix J). While this module has many uses, it has historically mainly been used to allow Windows based SIMPL modules to participate in a SIMPL application. The function API is similar to the `csimpl` API when applicable.

Suppose that you have a SIMPL application that is partially distributed on a Windows host. Suppose that the balance of this SIMPL application is deployed on a Linux host. The ‘C’ extended module `csimpl` is *NOT* compatible with Windows and the same will apply to any of SIMPL’s standard surrogates (TCP/IP and RS-232). The tclSurrogate approach divides the SIMPL module into two parts which communicate with each other using the TCP/IP tclSurrogate protocol. Typically the generic half is deployed on a Linux node. The business logic half is then free to be deployed elsewhere, typically on a Windows node. There is at this time no SIMPL version that allows Windows-Windows communications.

This module can be used to distribute a Linux only SIMPL application across multiple nodes. However, in that case the `csimpl` module is a better choice.

If the SIMPL application is Linux based but includes a deeply embedded node such as Arduino, then this `tclsimpl` module is needed. The SIMPL library that enables *Arduino* microcontroller boards to SIMPL communicate is based on the *tclSurrogate* (See Appendix J) code written in ‘C’.

The name *tclsimpl* comes from *tclSurrogate simpl*.

### E.1.3 Module Summary Table

Table E.1 summarizes all of the currently available Python SIMPL modules.

Package	Description	Linux/Unix/etc.	Windows
<code>csimpl</code>	‘C’ extensions support	yes	no
<code>tclsimpl</code>	tclSurrogate support	yes	yes

Table E.1: Summary of Python Modules

## E.2 Main Functionality

The main functionality is comprised of the functions/methods within the csimpl and telsimpl modules that are used for the sending and receiving of messages.

### E.2.1 Main Function Synopsis

```
integer nameAttach(name, maxMemSize)
```

*Supported Modules:* csimpl

*Description:* The nameAttach() function/method enables a program to participate in SIMPL communications; as such, it must always precede any other SIMPL calls. In the case that an instance of the csimpl class is made, nameAttach() is called as part of the class constructor.

The parameter *name* is a character string. The name is a unique SIMPL name for the calling process; that is, the it must be the only instance of that name in a given SIMPL sandbox.

The parameter *maxMemSize* is an integer which determines the size of the message buffers set within the csimpl module. This will determine the upper limit on the physical size of the incoming and outgoing messages, hence it is important that this number be large enough to accommodate any anticipated message sizes.

*Example 1:* This example name attaches the name "RECEIVER" and allocates the message buffer sizes to be 1024 bytes. The csimpl module is imported and nameAttach() is called as a function.

```
import csimpl
import sys

name = "RECEIVER"
retVal = csimpl.nameAttach(name, 1024)
if retVal == -1:
    sys.exit(-1)
```

*Example 2:* This is the same example as above but this time nameAttach() is called as part of the constructor when an instance of the Simpl class is made.

```
import csimpl
import sys
```

```

name = "RECEIVER"
# attach the SIMPL name via the instance nee's constructor
nee = csimpl.Simpl(name, 1024)
if nee == -1:
    sys.exit(-1)

```

**integer nameAttach(name)**

*Supported Modules:* tclsimpl

*Description:* tclsimpl is the imported module and the inference is that tclSurrogate remote communications are being used and this process resides on a Windows host. The parameter *name* then needs to convey more information. This is of the form *port:host:name* which is also a string. Port is the port being used for the remote communication, host is the name of the remote host computer (which is running Linux/Unix/etc.), and name is the unique SIMPL name of the calling process. For more on the tclSurrogate see Chapter 7 and Appendix J.

Note that the parameter maxMemSize is not included because the message buffers are set within the tclsimpl module.

*Example:* This process is running on a Windows host. It is required that it SIMPL communicates with another process on a Linux host. The tclsimpl module is imported to enable tclSurrogate remote communications. The port number is 50000, the IP of the remote host is 192.168.1.42, and the unique SIMPL name of the calling process is RECEIVER.

```

import tclsimpl
import sys

name = "" 50000:192.168.1.42:RECEIVER"
retVal = tclsimpl.nameAttach(name)
if retVal == -1:
    sys.exit(-1)

```

**integer nameDetach()**

*Supported Modules:* csimpl/tclsimpl

*Description:* The nameDetach() function/method releases all SIMPL functionality from a program.

*Example 1:* This script SIMPL name attached the name "RECEIVER" and no

longer requires SIMPL capability. The csimpl module is the source of the SIMPL functionality in this case.

```
import csimpl
import sys

name = "RECEIVER"
retVal = csimpl.nameAttach(name, 1024)
if retVal == -1:
    sys.exit(-1)
"""
    processing ...
"""
# detach name
csimpl.nameDetach()
```

*Example 2:* This script SIMPL name attached the name "RECEIVER" and no longer requires SIMPL capability. In the case that an instance of SIMPL being created, the nameDetach() is called via the object's destructor upon exit.

```
import csimpl
import sys

name = "RECEIVER"
# attach SIMPL name via the object nee's constructor
nee = csimpl.SIMPL(name, 1024)
if nee == -1:
    sys.exit(-1)
"""
    processing ...
"""
# SIMPL name detached at exit via the object nee's destructor
```

**integer surDetach(id)**

*Supported Modules:* csimpl

*Description:* The surDetach() function/method releases a specific surrogate process identified by the integer *id*. This function is not required by the tclsimpl module because the tclSurrogate methodology works differently from the TCP/IP and RS232 surrogate systems.

*Example:* This script has name attached the name "SENDER" and has name located a process called "RECEIVER" on a remote host called "remoteHost".

The sender no longer requires the connection to the receiver.

```
import csimpl
import sys

myName = "SENDER"
name = "remoteHost:RECEIVER"

# create an instance, name attach() is automatic
mySimp = csimpl.Simpl(myName, 1024)

# name locate remote process
remoteId = mySimp.nameLocate(name)
if remoteId == -1:
    sys.exit(-1)
"""
    processing, sending message(s) to "remoteHost:RECEIVER" ...
"""

# no longer need a SIMPL connection to this remote process
mySimp.surDetach(remoteId)
```

**integer nameLocate(name)**

*Supported Modules:* csimpl/tclsimpl

*Description:* The nameLocate() function/method is used by one SIMPL program to SIMPL connect to another SIMPL program for the purpose of sending messages. The parameter *name* is a string which may merely represent a unique SIMPL name or in the case of csimpl may be of the form: *protocol:host:SIMPL name* for remote message sending. See the description of name\_locate() in Appendix C.

*Example 1:* This script has name attached the name "SENDER" and then creates a SIMPL connection to a local (ie. same host) receiver called "RECEIVER".

```
import csimpl
import sys

myName = "SENDER"
recvName = "RECEIVER"

# SIMPL name attach
```

```

retVal = csimpl.nameAttach(myName, 1024)
if retVal == -1:
    sys.exit(-1)

# name locate "RECEIVER"
receiverId = csimpl.nameLocate(recvName)
if receiverId == -1:
    sys.exit(-1)
"""
        processing ...
"""

```

*Example 2:* This script has name attached the name "SENDER" and then creates a SIMPL connection to a remote receiver called "RECEIVER" using the RS-232 protocol on host called "remoteHost".

```

import csimpl
import sys
myName = "SENDER"
recvName = "RS232:remoteHost:RECEIVER"

# SIMPL name attach
retVal = csimpl.nameAttach(myName)
if retVal == -1:
    sys.exit(-1)

# name locate "RECEIVER"
receiverId = csimpl.nameLocate(recvName)
if receiverId == -1:
    sys.exit(-1)
"""
        processing ...
"""

```

**integer integer receive()**

*Supported Modules:* csimpl/tclsimpl

*Description:* The receive() call accepts messages from SIMPL senders. In the case of csimpl, the receive() function syntax for Python is somewhat different from its 'C' root function as examples 2. and 3. below will show. Given that Python prefers to manage memory dynamically, instead of creating memory buffers and passing them to Receive(), the Python receive() returns values as

they are set within the function. For more information see the description of Receive() in the ‘C’ language section.

*Example 1:* The following example demonstrates how a Python receiver accepts a message from a sender and forms a reply. The tclsimpl module is used in conjunction with the tclSurrogate indicating that the receiver would likely be running on a Windows host and SIMPL communicating with a non-Windows host based sender. The non-Windows host is a 64-bit computer. Note the use of the packing/unpacking functions with respect to the message content. The message format is character-based as opposed to binary-based.

```
from tclsimpl import *
from array import array
import sys

def constructMsg():
    packChar("A", SIMPL_CHR)
    packDouble(3.1415, SIMPL_CHR)
    packShort(2, SIMPL_CHR)
    packChar("B", SIMPL_CHR)
    packLong(24, SIMPL_CHR)
    packChar("C", SIMPL_CHR)
    packFloat(2.1718, SIMPL_CHR)
    packInt(37, SIMPL_CHR)

    a = array("c", "ZZZZZZZZ")
    b = array("h", [0, 1, 2, 3, 4, 5, 6, 7, 8])
    c = array("i", [0, 1, 2, 3, 4, 5, 6, 7, 8])
    d = array("l", [0, 1, 2, 3, 4, 5, 6, 7, 8])
    e = array("f", [1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9])
    f = array("d", [.1, .02, .003, .0004, .00005, .000006, 0.0000007])

    packCharArray(a, SIMPL_CHR)
    packShortArray(b, SIMPL_CHR)
    packIntArray(c, SIMPL_CHR)
    packLongArray(d, SIMPL_CHR)
    packFloatArray(e, SIMPL_CHR)
    packDoubleArray(f, SIMPL_CHR)

def deconstructMsg():
    print("receiver incoming message")
    print(unpackChar(SIMPL_CHR))
    print(unpackDouble(SIMPL_CHR))
    print(unpackShort(SIMPL_CHR))
    print(unpackChar(SIMPL_CHR))
    print(unpackLong(SIMPL_CHR))
    print(unpackChar(SIMPL_CHR))
```

```
print(unpackFloat(SIMPL_CHR))
print(unpackInt(SIMPL_CHR))

var1 = " " * 9
g = array("c", var1)
unpackCharArray(g, SIMPL_CHR)
print(g)

var2 = [0] * 9
h = array("h", var2)
unpackShortArray(h, SIMPL_CHR)
print(h)

var3 = [0] * 9
i = array("i", var3)
unpackIntArray(i, SIMPL_CHR)
print(i)

var4 = [0] * 9
j = array("l", var4)
unpackLongArray(j, SIMPL_CHR)
print(j)

var5 = [0] * 9
k = array("f", var5)
unpackFloatArray(k, SIMPL_CHR)
print(k)

var6 = [0] * 7
l = array("d", var6)
unpackDoubleArray(l, SIMPL_CHR)
print(l)

***** main part of program *****

# name attach
myName = "RECEIVER"
name = "50000:localhost:" + myName
ret = nameAttach(name, 1024)
if ret == -1:
    print("%s: name attach error" %(myName))
    sys.exit(-1)

# set message packing/unpacking scheme based on sender's architecture
setInArch(B64) # for the incoming message
setOutArch(B64) # for the outgoing reply
```

```

while 1:
    # receive an incoming message
    messageSize, senderId = receive()
    if messageSize == -1:
        # error
        print("%s: receive error" %(myName))
        sys.exit(-1)

    # examine the incoming message and print to screen
    deconstructMsg()

    # build the outgoing reply message
    constructMsg()

    # reply to sending program
    retVal = reply(senderId)
    if retVal == -1:
        print("%s: reply error" %(myName))
        sys.exit(-1)

```

*Example 2:* This script has name attached the name "RECEIVER" and then waits to receive messages from senders, then replying with a set number.

```

from csimpl import *
import sys

myName = "RECEIVER"

# SIMPL name attach
retVal = nameAttach(myName)
if retVal == -1:
    sys.exit(-1)

# receive a message
messageSize, senderId = receive()

# check for error
if messageSize == -1:
    print("%s: receive error-%s" %(myName, whatsMyError()))
    sys.exit(-1)
# is it a non-null message?
elif messageSize > 0:
    # extract the token from the message
    token = unpackInt(SIMPL_BIN)
    # react to the token

```

```

"""
    processing ...
"""

# reply to the sender
var = 7
packInt(var, SIMPL_BIN)
retVal = reply(senderId)
if retVal == -1:
    print("%s: reply error-%s" %(myName, whatsMyError()))
    sys.exit(-1)

```

*Example 3:* This script has name attached the name "RECEIVER" and then waits to receive proxies from senders. Note that there is no reply() required to a proxy. Note that proxies are supported by the csimpl module only, not by the tclsimpl module.

```

from csimpl import *
import sys
import struct

myName = "RECEIVER"

# SIMPL name attach
retVal = nameAttach(myName)
if retVal == -1:
    sys.exit(-1)

# receive a message
messageSize, senderId = receive()

# check for error
if messageSize == -1:
    print("%s: receive error-%s" %(myName, whatsMyError()))
    sys.exit(-1)
# is it a proxy?
elif messageSize < -1:
    proxy = returnProxy(messageSize)
    # react to the proxy value
"""

    processing ...
"""

```

**integer/none reply(integer)**

*Supported Modules:* csimpl/tclsimpl

*Description:* The reply() function responds to a sender after a receive(). In the case of csimpl, it returns a 0 for success and a -1 for failure. It returns nothing in the case of tclsimpl.

*Example 1:* This script has name attached the name "RECEIVER", waits to receive messages from senders, then replying with "Thanks".

```
from csimpl import *
import sys
import struct

myName = "RECEIVER"

# SIMPL name attach
retVal = nameAttach(myName)
if retVal == -1:
    sys.exit(-1)

# receive a message
messageSize, senderId = receive()
"""
        processing message ...
"""

# reply to the sender
var = "Thanks"
out = packMsg("s", var)
retVal = reply(senderId)
if retVal == -1:
    print("%s: reply error-%s" %(myName, whatsMyError()))
    sys.exit(-1)
```

*Example 2:* This script has name attached the name "RECEIVER" and then waits to receive messages from senders, then replying with nothing.

```
from csimpl import *
import sys

myName = "RECEIVER"

# SIMPL name attach
retVal = nameAttach(myName)
```

```

if retVal == -1:
    sys.exit(-1)

# receive a message
messageSize, senderId = receive()
"""
        processing message ...
"""

# reply to the sender
retVal = reply(senderId)
if retVal == -1:
    print("%s: reply error-%s" %(myName, whatsMyError()))
    sys.exit(-1)

```

### **integer send(integer)**

*Supported Modules:* csimpl/tclsimpl

*Description:* The send() call sends a message that is preconstructed using the packing functions and receives a reply from the target receiver. It returns the size of the replied message or -1 for a failure.

*Example:* This script has name attached the name "SENDER" and has name located a process called "RECEIVER". It creates a message via constructMsg(), sends the message to RECEIVER, collects the reply from RECEIVER and then examines the message within deconstructMsg(). This example uses csimpl as the source of the SIMPL functionality.

```

import csimpl
import sys

# make a simpl class object
myName = "SENDER"
bob = csimpl.Simpl(myName, 1024)

# name locate receiver program
id = bob.nameLocate("RECEIVER")
if id == -1:
    print("%s: name locate error-%s" %(myName, bob.whatsMyError()))
    sys.exit(-1)

# build outgoing message
constructMsg(bob)

```

```
# send the message to the receiver program
ret = bob.send(id)
if ret == -1:
    print("%s: send error-%s" %(myName, bob.whatsMyError()))
    sys.exit(-1)

# examine the return message, if any
deconstructMsg(bob)
```

### **integer trigger(integer, integer)**

*Supported Modules:* csimpl

*Description:* The trigger() call sends an unblocked (ie. expects no reply) proxy to the target receiver. It returns a 0 for success and a -1 for failure.

*Example:* The following example demonstrates how a Python sender triggers a receiver.

```
import csimpl
import sys

# initialize some required variables
sName = "SENDER"
rName = "RECEIVER"

# make a proxy value to send
proxy = 42

# attach a simpl name
retVal = csimpl.nameAttach(sName, 1024)
if retVal == -1:
    print("%s: name attach error-%s" %(sName, csimpl.whatsMyError()))
    sys.exit(-1)

# name locate the receiver
receiverId = csimpl.nameLocate(rName)
if receiverId == -1:
    print("%s: name locate error-%s" %(sName, csimpl.whatsMyError()))
    sys.exit(-1)

# generate a trigger
retVal = csimpl.trigger(receiverId, proxy)
if retVal == -1:
```

```
print("%s: trigger error-%s" %(sName, csimpl.whatsMyError()))
sys.exit(-1)

# exit gracefully
csimpl.nameDetach()
```

**integer returnProxy(integer)**

*Supported Modules:* csimpl

*Description:* The returnProxy() function syntax for Python is the same as the ‘C’ function call. For more information see the description of returnProxy() in the ‘C’ language section. Note that returnProxy() is not supported in tclsimpl because the tclSurrogate mechanism does not support triggers.

*Example:* See example 3 for the csimpl receive() for the sort of program that receives a proxy sent by the trigger() and has it converted to its true value.

**string whatsMyError()**

*Supported Modules:* csimpl

*Description:* The whatsMyError() function syntax for Python is comparable to the ‘C’ function call. For more information see the description of whatsMyError() in the ‘C’ language section.

*Example:* Its use can be found in many of the preceding examples.

**integer whatsMyReceiveFd()**

*Supported Modules:* csimpl

*Description:* The whatsMyReceiveFd() function syntax for Python is comparable to the ‘C’ function call. For more information see the description of whatsMyRecvfd() in the ‘C’ language section. It is a very important function because it allows the Python program to be made aware of an incoming message while it is perhaps waiting for input from some other source.

*Example:* A good example can be taken from a Python script that puts up a Tk front end expecting user input. How does this program know to run a receive() call in the case of an incoming message? See below.

```
import sys
from Tkinter import *
from csimpl import *

# define functionality to be performed when a message is received
def hndlMessage(a, b):
    # receive a message
    messageSize, senderId = receive()

    # process message ...

    # reply to the sender
    retVal = reply(senderId)
    if retVal == -1:
        print("%s: reply error-%s" %(rName, whatsMyError()))
        sys.exit(-1)

# main part of this program

# set this program's name
rName = "RECEIVER"

# attach a simpl name
retVal = nameAttach(rName)
if retVal == -1:
    print("%s: name attach error-%s" %(rName, whatsMyError()))
    sys.exit(-1)

# get the receive FIFO file descriptor
fd = whatsMyReceiveFd()

# initialize Tk for graphics
root = Tk()

# attach a callback for incoming simpl messages
root.tk.createfilehandler(fd, READABLE, hndlMessage)

# build user interface ...

# handle user input and simpl messaging
root.mainloop()
```

### E.2.2 Main Function Summary Table

Table E.2 summarizes all of the currently available Python SIMPL functions/methods and their respective modules.<sup>1</sup>

Function/Method	Purpose	Module
Simpl()	Simpl class constructor	c/t
integer nameAttach(string)	Start SIMPL functionality	t
integer nameAttach(string, integer)		c
integer nameDetach()	End SIMPL functionality	c/t
integer surDetach(integer)	Dismiss a surrogate	c
integer nameLocate(string)	Make a SIMPL connection	c/t
integer integer receive()	Receive a message	c/t
integer send(integer)	Send a message	c/t
integer reply(integer)	Reply to a received message	c/t
integer trigger(integer, integer)	Send a proxy	c
integer whatsMyReceiveFd()	Return receive fifo fd	c
string whatsMyError()	Return SIMPL error	c
integer returnProxy(integer)	Return proxy value	c

Table E.2: Summary of Python Main SIMPL Functions

## E.3 Packing and Unpacking Functions

The packing and unpacking functions/methods are those within the csimpl and tclsimpl modules that are responsible for composing and decomposing messages respectively.

Sending and receiving messages via SIMPL can be thought of in terms of:

1. Actions which construct and deconstruct the messages.
2. Actions which transfer these messages to and fro.

The packing/unpacking functions pertain to 1.

### E.3.1 Packing/Unpacking Function Synopsis

Packing and unpacking in Python is done with the aid of the *struct* module. This allows a Python program to compose messages that conform to ‘C’ data

---

<sup>1</sup>Table Modules: c=csimpl t=tclsimpl

types, hence the inclusion of functions that pack/unpack shorts, ints, floats etc. In Python, integers are ‘C’ long ints, floats are ‘C’ doubles, and dictionaries, for example, have no ‘C’ counterpart (enums, sort of). It is evident that Python takes a very different approach to basic data types than that of ‘C’ or Java. We use the same packing/unpacking API as that of Java for the sake of universality with the understanding that one must be clear what is meant by each data type, ie. what sort of program is sending it and what sort of program is receiving it. The ‘C’ data types have been chosen as a standard for the data types because Python scripts often send and receive messages from ‘C’ programs in our experience. As well, ‘C’ was the first language used to construct a SIMPL library and that is the basis for ‘C’ extensions used in the Python module *csimpl*.

### E.3.2 Packing Functions

The packing functions generally take the form *packType(val)* for primitives and *packTypeArray(val[])* for arrays. The type is variable primitive such as char, short, float etc. Val is the actual value of the primitive type. The datum is placed at that first available position within the outgoing message.

All of the packing functions write their particular datum or data into a memory buffer. In this case, it is referred to as *outBuffer*. The contents of *outBuffer* compose a message which is going to be sent to another SIMPL process, either by a *Send()* or by a *Reply()*.

It is possible to pack a message into the *outBuffer* without using the so-called packing functions. The *struct* module provides much functionality for positing various ‘C’ data types into a memory location. In this case outgoing message memory can be accessed via a variable called *\_simplOutBuf*. See the example at the end of this section.

### E.3.3 Unpacking Functions

The unpacking functions generally take the form *val = unpackType()* for primitives and *val[] = unpackTypeArray()* for arrays. The type is the variable primitive. Val is the actual value of the variable. The datum is retrieved from the next available position within the incoming message. It is very important that the message structure (datum order, size, byte order etc.) is of a form agreed upon by sender and receiver.

The byte, char, short, int, long, float and double unpacking methods return the value of the data type.

All of the unpacking functions read their particular datum or data from a memory buffer. In this case, it is referred to as inBuffer. The contents of inBuffer compose a message which has been received from another SIMPL process, either by a Receive() or by the reply to a Send().

It is also possible to unpack a message from the inBuffer without using the so-called unpacking functions. The *struct* module provides much functionality for extracting various ‘C’ data types from a memory location. In this case incoming message memory can be accessed via a variable called *\_simlInBuf*. Again, see the example at the end of this section.

### E.3.4 Example 1

For our example a Python sender and a Python receiver have been chosen. It can be assumed that both programs are local to the same host (note the syntax of the sender’s nameLocate()). A message with all of the currently supported data types is used for both sending and replying in order to provide an example of the usage of all available packing/unpacking functions. The receiver program constructs the same reply message as the message it has received. Both programs have imported the *csimpl* module. Given that this module is based on ‘C’ extensions it will only run on Unix, Linux, MAC OS X etc. We have purposely set the sent and replied messages to be the same; repetition is a good teacher.

#### sender.py

```
1 #import required modules
2 import csimpl
3 from array import array
4 import sys
5
6 def constructSendMsg(nee):
7     # create arrays of all supported types for an outgoing message
8     a = array("b", [True, False, True, False, True, False])
9     b = array("h", [0, 1, 2, 3, 4, 5, 6, 7, 8])
10    c = array("i", [0, 1, 2, 3, 4, 5, 6, 7, 8])
11    d = array("l", [0, 1, 2, 3, 4, 5, 6, 7, 8])
12    e = array("f", [1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9])
13    f = array("d", [.1, .02, .003, .0004, .00005, .000006, 0.0000007])
14
15    # pack the outgoing message buffer
16    nee.packBool(True, nee.SIMPL_BIN)
17    nee.packChar("A", nee.SIMPL_BIN)
18    nee.packShort(2, nee.SIMPL_BIN)
19    nee.packInt(37, nee.SIMPL_BIN)
```

```
20     nee.packLong(42424242, nee.SIMPL_BIN)
21     nee.packFloat(2.1718, nee.SIMPL_BIN)
22     nee.packDouble(3.1415, nee.SIMPL_BIN)
23     nee.packString("We are the knights who say nee!", nee.SIMPL_BIN)
24     nee.packBoolArray(a, nee.SIMPL_BIN)
25     nee.packShortArray(b, nee.SIMPL_BIN)
26     nee.packIntArray(c, nee.SIMPL_BIN)
27     nee.packLongArray(d, nee.SIMPL_BIN)
28     nee.packFloatArray(e, nee.SIMPL_BIN)
29     nee.packDoubleArray(f, nee.SIMPL_BIN)
30
31 def deconstructReplyMsg(nee):
32     # extract and print the contents of the incoming message
33     print(nee.unpackBool(nee.SIMPL_BIN))
34     print(nee.unpackChar(nee.SIMPL_BIN))
35     print(nee.unpackShort(nee.SIMPL_BIN))
36     print(nee.unpackInt(nee.SIMPL_BIN))
37     print(nee.unpackLong(nee.SIMPL_BIN))
38     print(nee.unpackFloat(nee.SIMPL_BIN))
39     print(nee.unpackDouble(nee.SIMPL_BIN))
40     print(nee.unpackString(31, nee.SIMPL_BIN))
41     print(nee.unpackBoolArray(6, nee.SIMPL_BIN))
42     print(nee.unpackShortArray(9, nee.SIMPL_BIN))
43     print(nee.unpackIntArray(9, nee.SIMPL_BIN))
44     print(nee.unpackLongArray(9, nee.SIMPL_BIN))
45     print(nee.unpackFloatArray(9, nee.SIMPL_BIN))
46     print(nee.unpackDoubleArray(7, nee.SIMPL_BIN))
47
48 ##### main part of program ****
49
50 # create an instance of Simpl
51 nee = csimpl.Simpl("PACK_ALL_POSSIBLE_TYPES_S", 1024)
52
53 # name locate receiver program
54 nid = nee.nameLocate("PACK_ALL_POSSIBLE_TYPES_R")
55 if nid == -1:
56     print(myName, ": name locate error-", nee.whatsMyError())
57     sys.exit(-1)
58
59 # build outgoing message
60 constructSendMsg(nee)
61
62 # send the message to the receiver program
63 ret = nee.send(nid)
64 if ret == -1:
65     print(myName, ": send error-", nee.whatsMyError())
```

```
66     sys.exit(-1)
67
68 # examine and display the reply message
69 deconstructReplyMsg(nee)
```

lines 2-4 Import required modules.

lines 6-29 A function for constructing the message to be sent to the receiver via the individual packing methods.

lines 31-46 A function for deconstructing the reply message from the receiver via the individual unpacking methods and printing out the contents.

line 51 An instance of the Simpl object is created. The required SIMPL name attach is made by the constructor. Note the argument 1024 (1 kbyte). This is to set the maximum size of the message buffer used for sending messages and is required by the ‘C’ extension used by the csimpl module. Note that setting the upper limit for the message size is unnecessary when using the tclsimpl module because that module is entirely written in Python.

lines 54-57 The SIMPL name locate for the receiver program is made. This returns an identification number for the particular receiver that goes by the name *PACKi\_ALi\_POSSIBLE\_TYPES\_R* and is used by the Send() for sending the message to the desired place.

line 60 Construct the message to be sent.

lines 63-66 Send the message to the receiver and get the reply message from the receiver.

line 69 Deconstruct and print out the reply message from the receiver.

*receiver.py*

```
1 #import required modules
2 import csimpl
3 from array import array
4 from sys import exit
5
6 def constructReplyMsg(nee):
7     # arrays of all supported types for an outgoing message
8     a = array("b", [True, False, True, False, True, False])
9     b = array("h", [0, 1, 2, 3, 4, 5, 6, 7, 8])
10    c = array("i", [0, 1, 2, 3, 4, 5, 6, 7, 8])
```

```

11 d = array("l", [0, 1, 2, 3, 4, 5, 6, 7, 8])
12 e = array("f", [1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9])
13 f = array("d", [.1, .02, .003, .0004, .00005, .000006, 0.0000007])
14
15 # pack the outgoing message buffer
16 nee.packBool(True, nee.SIMPL_BIN)
17 nee.packChar("A", nee.SIMPL_BIN)
18 nee.packShort(2, nee.SIMPL_BIN)
19 nee.packInt(37, nee.SIMPL_BIN)
20 nee.packLong(42424242, nee.SIMPL_BIN)
21 nee.packFloat(2.1718, nee.SIMPL_BIN)
22 nee.packDouble(3.1415, nee.SIMPL_BIN)
23 nee.packString("We are the knights who say nee!", nee.SIMPL_BIN)
24 nee.packBoolArray(a, nee.SIMPL_BIN)
25 nee.packShortArray(b, nee.SIMPL_BIN)
26 nee.packIntArray(c, nee.SIMPL_BIN)
27 nee.packLongArray(d, nee.SIMPL_BIN)
28 nee.packFloatArray(e, nee.SIMPL_BIN)
29 nee.packDoubleArray(f, nee.SIMPL_BIN)
30
31 def deconstructReceiveMsg(nee):
32     # extract and print the contents of the incoming message
33     print(nee.unpackBool(nee.SIMPL_BIN))
34     print(nee.unpackChar(nee.SIMPL_BIN))
35     print(nee.unpackShort(nee.SIMPL_BIN))
36     print(nee.unpackInt(nee.SIMPL_BIN))
37     print(nee.unpackLong(nee.SIMPL_BIN))
38     print(nee.unpackFloat(nee.SIMPL_BIN))
39     print(nee.unpackDouble(nee.SIMPL_BIN))
40     print(nee.unpackString(31, nee.SIMPL_BIN))
41     print(nee.unpackBoolArray(6, nee.SIMPL_BIN))
42     print(nee.unpackShortArray(9, nee.SIMPL_BIN))
43     print(nee.unpackIntArray(9, nee.SIMPL_BIN))
44     print(nee.unpackLongArray(9, nee.SIMPL_BIN))
45     print(nee.unpackFloatArray(9, nee.SIMPL_BIN))
46     print(nee.unpackDoubleArray(7, nee.SIMPL_BIN))
47
48 ##### main part of program ****
49
50 # create an instance of Simpl
51 nee = csimpl.Simpl("PACK_ALL_POSSIBLE_TYPES.R", 1024)
52
53 while True:
54     # receive a message
55     messageSize, senderId = nee.receive()
56     if messageSize == -1:

```

```
57     print(" receive error-", nee.whatsMyError())
58     exit(-1)
59
60     # examine the message
61     deconstructReceiveMsg(nee)
62
63     # build reply message
64     constructReplyMsg(nee)
65
66     # reply to sending program
67     retVal = nee.reply(senderId)
68     if retVal == -1:
69         print(" reply error-%s", nee.whatsMyError())
70         exit(-1)
```

lines 2-4 Import required modules.

lines 6-29 A function for constructing the reply message to be sent to the sender via the individual packing methods.

lines 31-46 A function for deconstructing the sent message from the sender via the individual unpacking methods and printing out the contents.

line 51 An instance of the Simpl object is created. The required SIMPL name attach is made by the constructor. Note the argument 1024 (1 kbyte). This is to set the maximum size of the message buffer used for sending messages and is required by the ‘C’ extension used by the csimpl module. Note that setting the upper limit for the message size is unnecessary when using the tclsimpl module because that module is entirely written in Python.

lines 53-70 The program loops endlessly receiving messages.

lines 55-58 The program receives incoming messages.

line 61 The program deconstructs the received message from the sender and prints out the contents.

line 64 The program constructs the reply message to be sent back to the sender.

lines 67-70 The program replies its message back to the sender.

### E.3.5 Example 2

This example is identical to the first example with one exception: Instead of using the packBool()/unpackBool etc. methods, we use the more general packMsg()/unpackMsg() methods which use format styles similar to the ‘C’ printf()/scanf() functions respectively. The general method can be preferable to using the individual methods in the case of smaller messages. In the case of larger and more complex messages however, it might be more readable to use the individual methods.

#### *sender.py*

```

1 #import required modules
2 import csimpl
3 from array import array
4 import sys
5
6 def constructSendMsg(nee):
7     # create arrays of all supported types for an outgoing message
8     a = array("b", [True, False, True, False, True, False])
9     b = array("h", [0, 1, 2, 3, 4, 5, 6, 7, 8])
10    c = array("i", [0, 1, 2, 3, 4, 5, 6, 7, 8])
11    d = array("l", [0, 1, 2, 3, 4, 5, 6, 7, 8])
12    e = array("f", [1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9])
13    f = array("d", [.1, .02, .003, .0004, .00005, .000006, 0.0000007])
14
15    # pack the outgoing message buffer
16    nee.packMsg(nee.SIMPL_BIN, "bchilfd3s6B9H9I9L9F7D", True, "A", 2, 37,
17        42424242, 2.1718, 3.1415, "we are the knights who say nee!",
18        a, b, c, d, e, f)
19
20 def deconstructReplyMsg(nee):
21     # extract the reply message
22     h,i,j,k,l,m,n,o,p,q,r,s,t,u =
23         nee.unpackMsg(nee.SIMPL_BIN, "bchilfd3s6B9H9I9L9F7D")
24
25     # print the reply msg contents
26     print(h)
27     print(i)
28     print(j)
29     print(k)
30     print(l)
31     print(m)
32     print(n)
33     print(o)
34     print(p)
```

```
35     print(q)
36     print(r)
37     print(s)
38     print(t)
39     print(u)
40
41 #***** main part of program *****
42
43 # create an instance of Simpl
44 nee = csimpl.Simpl("PACK_ALL_POSSIBLE_TYPES_S", 1024)
45
46 # name locate receiver program
47 nid = nee.nameLocate("PACK_ALL_POSSIBLE_TYPES_R")
48 if nid == -1:
49     print(myName, ": name locate error-", nee.whatsMyError())
50     sys.exit(-1)
51
52 # build outgoing message
53 constructSendMsg(nee)
54
55 # send the message to the receiver program
56 ret = nee.send(nid)
57 if ret == -1:
58     print(myName, ": send error-", nee.whatsMyError())
59     sys.exit(-1)
60
61 # examine and display the reply message
62 deconstructReplyMsg(nee)
```

lines 2-4 Import required modules.

lines 6-18 A function for constructing the message to be sent to the receiver via the general packing method.

lines 20-23 A function for deconstructing the reply message from the receiver via the general unpacking method and printing out the contents.

line 44 The SIMPL class instance is created.

lines 47-50 The SIMPL name locate for the receiver program is made.

line 53 Construct the message to be sent.

lines 56-59 Send the message to the receiver and get the reply message from the receiver.

line 62 Deconstruct the reply message from the receiver.

### *receiver.py*

```

1 #import required modules
2 import csimpl
3 from array import array
4 from sys import exit
5
6 def constructReplyMsg(nee): # arrays of all supported types
7     a = array("b", [True, False, True, False, True, False])
8     c = array("h", [0, 1, 2, 3, 4, 5, 6, 7, 8])
9     d = array("i", [0, 1, 2, 3, 4, 5, 6, 7, 8])
10    e = array("l", [0, 1, 2, 3, 4, 5, 6, 7, 8])
11    f = array("f", [1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9])
12    g = array("d", [.1, .02, .003, .0004, .00005, .000006, 0.0000007])
13
14    # pack the outgoing message buffer
15    nee.packMsg(nee.SIMPL_BIN, "bchilfdSBHILFD", True, "A", 2, 37,
16                42424242, 2.1718, 3.1415, "we are the knights who say nee!",
17                a, b, c, d, e, f, g)
18
19 def deconstructReceiveMsg(nee):
20    # extract the incoming message
21    h,i,j,k,l,m,n,o,p,q,r,s,t,u = \
22        nee.unpackMsg(nee.SIMPL_BIN, "bchilfd31s6B9H9I9L9F7D")
23
24    # print the reply msg contents
25    print(h)
26    print(i)
27    print(j)
28    print(k)
29    print(l)
30    print(m)
31    print(n)
32    print(o)
33    print(p)
34    print(q)
35    print(r)
36    print(s)
37    print(t)
38    print(u)
39
40 ##### main part of program ****
41
42 # create an instance of Simpl
43 nee = csimpl.Simpl("PACK_ALL_POSSIBLE_TYPES_R", 1024)

```

```
44
45 while True:
46     # receive a message
47     messageSize, senderId = nee.receive()
48     if messageSize == -1: # error
49         print("receive error-", nee.whatsMyError())
50         exit(-1)
51
52     # examine the message
53     deconstructReceiveMsg(nee)
54
55     # build reply message
56     constructReplyMsg(nee)
57
58     # reply to sending program
59     retVal = nee.reply(senderId)
60     if retVal == -1:
61         print("reply error-%s", nee.whatsMyError())
62         exit(-1)
```

lines 2-4 Import required modules.

lines 6-17 A function for constructing the reply message to be sent to the sender via the general packing method.

lines 19-38 A function for deconstructing the send message from the sender via the general unpacking method and printing out the contents.

line 43 The SIMPL class instance is created.

lines 45-62 The program loops endlessly receiving messages.

lines 47-50 The program receives incoming messages.

line 53 The program deconstructs the received message from the sender and prints out the contents.

line 56 The program constructs the reply message to be sent back to the sender.

lines 59-62 The program replies its message back to the sender.

### E.3.6 Packing/Unpacking Function Summary Table

Table E.3 summarizes all of the currently available Python SIMPL packing and unpacking functions/methods and their respective modules. The *type* refers to SIMPL\_BIN for binary-based and SIMPLi\_CHR for character-based messages.<sup>2</sup>

## E.4 Miscellaneous Functions

There are seven so-called miscellaneous functions. The first is called *crossCopy()*. It is normally used in the development side of the SIMPL software. It's only purpose is to copy the contents of the inBuffer message memory to the outBuffer message memory. This has been useful for testing the packing and unpacking methods and for verifying the contents of messages.

The next two, *returnInBuffer()* and *returnOutBuffer()* are concerned with returning the contents of the inBuffer and the outBuffer from receive()/reply() and send() respectively. That is, the inBuffer is the memory space that is loaded with an incoming message to a receive. It is also the same memory space that is used for the reply to a send. The outBuffer is the memory space loaded with the contents of a message which is to be sent to a receiver. It is also the memory space that is used by a reply for sending back messages to senders. These are the message buffers that are packed and unpacked by the pack/unpack functions normally.

The next function is called *loadOutBuffer(message)*. We saw it used in the previous section on packing and unpacking messages without the use of the normal packing and unpacking routines. It is used for putting *message* into the outBuffer prior to sending or replying.

The next two miscellaneous functions, *setInByteOrder()* and *setOutByteOrder()* are used to set the byte ordering on the incoming and outgoing messages and should be called prior to the pack/unpack function calls. The argument passed into these functions can be *SIMPL\_NATIVE*, *SIMPL\_BIG\_ENDIAN*, or *SIMPL\_LITTLE\_ENDIAN*. See Section 7.5.6.

Lastly, *setDMs()* is used to set the type of architecture for the message.

### E.4.1 Messages without Packing and Unpacking

Messages can be packed and unpacked without the use of the packing and unpacking functions. This can be done via the *struct* module and some of the

---

<sup>2</sup>Messages are either *packed* in a binary or a character format and must be *unpacked* likewise.

Packing Functions		
Function	Purpose	Return
packMsg(type a, form f, arg1, arg2, ...)	Put format f args into out buffer	nil
packBool(boolean p, type a)	Put boolean p into out buffer	nil
packChar(char p, type a)	Put char p into out buffer	nil
packShort(short p, type a)	Put short p into out buffer	nil
packInt(int p, type a)	Put int p into out buffer	nil
packLong(long p, type a)	Put long p into out buffer	nil
packFloat(float p, type a)	Put float p into out buffer	nil
packDouble(double p, type a)	Put double p into out buffer	nil
packString(char[ ] p, type a)	Put char array p into out buffer	nil
packBoolArray(bool[ ] p, type a)	Put boolean array p into out buffer	nil
packShortArray(short[ ] p, type a)	Put short array p into out buffer	nil
packIntArray(int[ ] p, type a)	Put int array p into out buffer	nil
packLongArray(long[ ] p, type a)	Put long array p into out buffer	nil
packFloatArray(float[ ] p, type a)	Put float array p into out buffer	nil
packDoubleArray(double[ ] p, type a)	Put double array p into out buffer	nil
Unpacking Functions		
Function	Purpose	Return
unpackMsg(type a, form f)	Get format f args from in buffer	args
unpackBool(type a)	Return boolean from in buffer	byte
unpackChar(type a)	Return char from in buffer	char
unpackShort(type a)	Return short integer from in buffer	short
unpackInt(type a)	Return integer from in buffer	int
unpackLong(type a)	Return long integer from in buffer	long
unpackFloat(type a)	Return float from in buffer	float
unpackDouble(type a)	Return double from in buffer	double
unpackString(type a)	Return char string from in buffer	string
unpackBoolArray(length i, type a)	Return boolean array from in buffer	boolean array
unpackShortArray(length i, type a)	Return short array from in buffer	short array
unpackIntArray(length i, type a)	Return int array from in buffer	int array
unpackLongArray(length i, type a)	Return long array from in buffer	long array
unpackFloatArray(length i, type a)	Return float array from in buffer	float array
unpackDoubleArray(length i, type a)	Return double array from in buffer	double array

Table E.3: Python SIMPL Packing/Unpacking Functions

miscellaneous functions for example. Suppose that you have some data type that is not readily covered by the available packing/unpacking functionality but you want to send or receive it as it is anyway. This would be the time to use the functionality outside of the packing/unpacking routines.

## E.4.2 Example 1

Observe the sender and receiver in the following example; neither one uses the available packing or unpacking routines to compose and decompose their messages.

*sender.py*

```

1 # import required modules
2 import sys
3 import struct
4 import csimpl
5
6 # initialize necessary variables
7 sName = "SENDER"
8 rName = "RECEIVER"
9
10 # enable this script to utilize simpl
11 retVal = csimpl.nameAttach(sName, 1024)
12 if retVal == -1:
13     print("%s: name attach error-%s" %(sName, csimpl.whatsMyError()))
14     sys.exit(-1)
15
16 # name locate the receiver program
17 receiverId = csimpl.nameLocate(rName)
18 if receiverId == -1:
19     print("%s: name locate error-%s" %(sName, csimpl.whatsMyError()))
20     sys.exit(-1)
21
22 # compose the message
23 outMsg = struct.pack("iii", 9, 99, 999)
24 csimpl.loadOutBuffer(outMsg)
25
26 # send the message
27 retVal = csimpl.send(receiverId)
28 if retVal == -1:
29     print("%s: send error-%s" %(sName, csimpl.whatsMyError()))
30     sys.exit(-1)
31 elif (retVal > 0):
32     inMsg = csimpl.returnInBuffer()
33     t = struct.unpack("i", inMsg[0:struct.calcsize("i")])
34     print(t[0])
35
36 csimpl.nameDetach()
```

lines 2-4 Import required modules.

lines 7-8 Define the unique SIMPL names for the sending and receiving programs.

lines 11-14 SIMPL name attach this sending process.

lines 17-20 SIMPL name locate the receiving process.

line 23 Build the message to be sent consisting of three integers.

line 24 Load the message into the outgoing message buffer.

lines 27-30 Send the message to the receiving process.

line 32 After a successful send, extract the replied message from the receiver.

lines 33-34 Extract a single integer from the replied message and print it to the screen.

line 36 Detach the SIMPL name from this process and end.

### receiver.py

```
1 # import required modules
2 import sys
3 import struct
4 import csimpl
5
6 # set this program's name
7 rName = "RECEIVER"
8 SINT = struct.calcsize("i")
9
10 # enable this script to utilize simpl
11 retVal = csimpl.nameAttach(rName, 1024)
12 if retVal == -1:
13     print("%s: name attach error-%s" %(rName, csimpl.whatsMyError()))
14     sys.exit(-1)
15
16 # receive a message
17 messageSize, senderId = csimpl.receive()
18 if messageSize == -1:
19     # error
20     print("%s: receive error-%s" %(rName, csimpl.whatsMyError()))
21     sys.exit(-1)
22 else:
23     # message
```

```
24     inMsg = csimpl.returnInBuffer()
25     t = struct.unpack("i", inMsg[0:SINT])
26     print(t[0])
27     t = struct.unpack("i", inMsg[SINT:2 * SINT])
28     print(t[0])
29     t = struct.unpack("i", inMsg[2 * SINT:3 * SINT])
30     print(t[0])
31
32     # reply to sending program
33     outMsg = struct.pack("i", 7)
34     csimpl.loadOutBuffer(outMsg)
35     retVal = csimpl.reply(senderId)
36     if retVal == -1:
37         print("%s: reply error-%s" %(rName, csimpl.whatsMyError()))
38         sys.exit(-1)
39
40     csimpl.nameDetach()
```

lines 2-4 Import required modules.

line 7 Define the unique SIMPL name for this receiving program.

line 8 Determine the size of an integer for this host.

lines 11-14 SIMPL name attach this receiving process.

lines 17-20 SIMPL name locate the receiving process.

line 17 Receive any incoming message.

lines 24-30 Unpack the incoming message into three integers.

line 33 Construct a response consisting of a single integer.

line 34 Load the response into the outgoing message buffer.

lines 35-38 Reply back to the sender.

line 40 Detach the SIMPL name from this process and end.

### E.4.3 Example 2

The following is an example which relies on more of the miscellaneous routines because it is a computer host sending to an Arduino Uno receiver.

receiver.c

```
1 #include <SPI.h>
2 #include <Ethernet.h>
3 #include <Simpl.h>
4
5 // structure definition of received/replied message
6 typedef struct
7 {
8     int num;
9 } MSG;
10
11 EthernetClient client;
12
13 void setup()
14 {
15     // port that the tclSurrogate is babysitting
16     const unsigned port = 8000;
17     // host where tclSurrogate lives
18     char *serverName = "www.icanprogram.ca";
19     // ethernet MAC address
20     byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x0D, 0x77};
21
22     // start serial messaging
23     Serial.begin(9600);
24     Serial.println();
25     Serial.println("starting up");
26
27     // start ethernet
28     if (!Ethernet.begin(mac))
29     {
30         Serial.println("error on ethernet");
31         while(1);
32     }
33     // give the ethernet shield a chance to initialize
34     delay(1000);
35
36     // remotely SIMPL name attach the name "ARDUINO"
37     if (nameAttach(serverName, port, &client, "ARDUINO") == -1)
38     {
39         Serial.println(whatsMyError());
40         client.stop();
```

```

41     while (1);
42 }
43
44 Serial.println("name attach OK");
45 Serial.println("initialization complete");
46 }
47
48 void loop()
49 {
50 long senderId; // the unique id of the sending program
51 MSG msg; // incoming/outgoing message variable
52 int ret; // function return value
53
54 // check for an incoming message
55 ret = Receive(&client, &senderId, &msg, sizeof(MSG));
56 if (ret == -1) // problem
57 {
58     Serial.println(whatsMyError());
59     nameDetach(&client);
60     while(1);
61 }
62 else if (ret > 0) // incoming message
63 {
64     // reply to the remote sender
65     msg.num *= 2;
66     if (Reply(&client, &senderId, &msg, sizeof(MSG) == -1)
67     {
68         Serial.println(whatsMyError());
69         nameDetach(&client);
70         while(1);
71     }
72 }
73 }
```

lines 1-3 Include required header files. Note line 3, that is the name of the header file for Arduino SIMPL.

lines 6-9 Define the receive and reply messages as a C structure with one element: an integer number.

lines 13-45 Arduino sketches usually start with a *setup* function that initializes functionality and variables.

lines 23-25 Enable this program to display messages via the USB connection to a host computer.

lines 28-34 Initialize the ethernet shield attached to the Uno.

lines 37-42 SIMPL name attach via the remote tclSurrogate.

lines 48-73 Arduino sketches usually have a function called *loop* that runs inside of an infinite loop.

lines 55-72 Receive messages of the type defined by the structure *MSG*, double the integer contained in the message and reply it back to the sender.

### sender.py

```
1 # import necessary modules
2 import sys
3 import tclsimpl
4
5 # make an instance of Simpl
6 nee = tclsimpl.Simpl("8000:www.icanprogram.ca:NUMSENDER")
7
8 # name locate the C program SIMPL receiver on the Arduino Uno
9 receiverId = nee.nameLocate("ARDUINO")
10 if receiverId == -1:
11     print("name attach error", ": is the receiver program running?")
12     sys.exit(-1)
13 """
14 The Arduino Uno uses a LP32 data model so set the necessary data
15 sizes on the outgoing messages so that the Uno can interpret the
16 messages correctly. As well, this host uses a LP64 data model, so
17 set the reply message from the Uno so that this sender can interpret
18 that message correctly.
19 """
20
21 For example, C ints are 4 bytes in LP64 and 2 bytes in LP32.
22 Consequently, ints must be sent by this sender as 2 bytes otherwise
23 they will be too large for the Uno to understand. On the reply side,
24 the ints coming back to this sender will only be 2 bytes in size and
25 must be interpreted accordingly.
26 """
27 nee.setDMs(nee.SIMPL_IN_DM, nee.SIMPL_LP32)
28 nee.setDMs(nee.SIMPL_OUT_DM, nee.SIMPL_LP32)
29
30 # Set the byte ordering such that it is compatible with the Arduino Uno.
31 nee.setInByteOrder(nee.SIMPL_BIG_ENDIAN)
32 nee.setOutByteOrder(nee.SIMPL_BIG_ENDIAN)
33
34 while True:
35     # get a number from keyboard entry
```

```
36 if sys.version > '3':
37     key = input("Enter a number ('q' to quit): ")
38 else:
39     key = raw_input("Enter a number ('q' to quit): ")
40 if key == 'q':
41     sys.exit(0)
42 num1 = int(key)
43
44 # load the message to be sent to the Arduino sketch
45 nee.packMsg(nee.SIMPL_BIN, "i", num1)
46
47 # send the message
48 if nee.send(receiverId) == -1:
49     print("send error")
50     sys.exit(-1)
51
52 num2 = nee.unpackMsg(nee.SIMPL_BIN, "i")
53 print("The original number is ", num1)
54 print("The doubled number is ", num2)
```

lines 2-3 Import the needed modules.

line 6 Create an instance of SIMPL.

lines 9-12 Name locate the SIMPL receiver running on the Arduino.

line 27 Set the incoming message buffer to data model LP32. This will make any messages replied from the receiver running on the Arduino compatible with the sender's host computer; ie. so the sender can read the replied message correctly.

line 28 Set the outgoing message buffer to data model LP32. This will make any messages sent to the receiver running on the Arduino compatible with the Arduino's data model; ie. so the receiver can read the incoming message correctly.

lines 31-32 Set the incoming and outgoing messages to be read and written using big endian byte ordering.

lines 34-54 Create messages to send to the Arduino receiver, get the reply and print to the screen.

line 45 Load the message into the outgoing message buffer in order to send it to the Arduino receiver.

line 48 Send the contents of the out buffer to the Arduino receiver.

line 52 Extract the replied message coming from the Arduino receiver from the incoming message buffer.

#### E.4.4 Miscellaneous Function Summary Table

Table E.4 summarizes all of the currently available Python SIMPL miscellaneous functions/methods which are available in both the csimpl and tclsimpl modules.

Function/Method	Purpose	Return
crossCopy()	Copy in buffer to out buffer (for testing)	nil
returnInBuffer()	Return the contents of inBuffer	binary
returnOutBuffer()	Return the contents of outBuffer	binary
loadOutBuffer(message)	Put message contents into out buffer	nil
setInByteOrder(order o)	Set the incoming message byte order	nil
setOutByteOrder(order o)	Set the outgoing message byte order	nil
setDMs(buffer x, arch y)	Set message buffer x to data model y	nil

Table E.4: Summary of Python SIMPL Miscellaneous Functions



# Appendix F

## Java Language Methods

### F.1 SIMPL Classes

There are two Java classes that offer SIMPL functionality. The reason for this is that each class was designed to address a specific need. The situation is similar to that of Python where there are two separate Python modules as well. This is because each *class* in the case of Java or *module* in the case of Python has been designed to SIMPL communicate over networks using either the tclSurrogate protocol *OR* one of the other more standard protocols.

With a few exceptions the methods contained in both of these classes have identical APIs. The two classes are called *Jsimpl* and *Ssimpl* respectively. We will briefly discuss both of these classes in what follows,

#### F.1.1 Jsimpl

*Jsimpl* contains SIMPL methods in a file called *Jsimpl.class*. *Jsimpl.class* refers to a shared Java Native Interface library called *libjsimpl.so* which provides the actual ‘C’-based SIMPL calls. This library is the one of choice when using SIMPL on standalone or networks of computers using non-Windows operating systems such as UNIX, Linux, MAC OS X etc.

#### F.1.2 Ssimpl

*Ssimpl* contains SIMPL methods in a file called *Ssimpl.class*. *Ssimpl* methods are written entirely in Java and make no reference to any other language libraries like *Jsimpl* does. This allows Windows-based hosts to SIMPL communicate with UNIX-based or Linux-based hosts in conjunction with the tclSurrogate. This is

*not* a method for Windows-based hosts to SIMPL communicate with each other. At this time, there is no completely Windows-based version of SIMPL.

### F.1.3 Class Summary Table

Table F.1 summarizes all of the currently available Java SIMPL classes.

Class Package	Description	Linux/Unix/etc.	Windows
Jimpl	Native Interface support	yes	no
Simpl	tclSurrogate support	yes	yes

Table F.1: Summary of Java Classes

## F.2 Main Method Synopsis

The following contains descriptions of the Java SIMPL methods offered at the time of printing.

**Jimpl() or Jimpl(int iSize, int oSize)**

**Simpl() or Simpl(int arch, String ord, int iSize, int oSize)**

*Description:* The Jimpl() and Simpl() constructors are required by every process that expects to send and/or receive SIMPL messages to and/or from any other process. They are responsible for setting up all of the SIMPL functionality that may be required. They must precede any other SIMPL library calls due to instantiation.

Jimpl() is the constructor for Jimpl that uses defaulted values for memory buffers. Any messages that are to be received or sent are buffered in memory. Both the receiving and sending buffers are set to 1024 bytes by default. Jimpl(int iSize, int oSize) allows the incoming and outgoing message buffers to be set as desired.

Simpl() is the constructor for Simpl that uses defaulted values for architecture type, byte ordering, and memory buffers. By default the architecture is 64-bit, the byte ordering is NATIVE to the host computer making the call, and both the receiving and sending buffers are set to 1024 bytes. Simpl(int arch, String

ord, int iSize, int oSize) allows the architecture to be set to either 64/32 bits, the byte order to be set to "NATIVE"/"LITTLE\_ENDIAN"/"BIG\_ENDIAN" as well as the incoming and outgoing message buffers to be set accordingly. The reasons for the architecture and byte ordering parameters is that the remote host may have different values of these parameters than the calling host. For example, if a 64-bit host wishes to send a message to a 32-bit host, then the outgoing message must be written for the 32-bit host. Any replied message must also be perceived as 32-bit and converted where necessary. So, the SIMPL instantiation might look like *Simpl simpl = Simpl(32, "NATIVE", 1024, 1024)*.

*Example One:* The following Jsimpl constructor uses the default values for the incoming/outgoing message buffers and creates an Jsimpl instance called "simpl".

```
class Example1
{
    public static void main( String arg [] )
    {
        // Jsimpl object constructor
        Jsimpl simpl = new Jsimpl();

        // SIMPL communications and processing here
    }
}
```

*Example Two:* The following Jsimpl constructor creates a Jsimpl instance called "simpl" and sets the size of the incoming message buffer to 512 bytes and the outgoing message buffer to 0. This would be allowed if there was no reply message expected.

```
class Example2
{
    public static void main( String arg [] )
    {
        // Jsimpl object constructor
        Jsimpl simpl = new Jsimpl(512, 0);

        // SIMPL communications and processing here
    }
}
```

*Example Three:* The following Ssimpl constructor creates an instance called "simpl" and sets the architecture to 32-bit, the byte-ordering to LITTLE\_ENDIAN and the sizes of the incoming and outgoing message buffers to 512 bytes.

```
class Example3
{
    public static void main( String arg [] )
    {
        // Ssimpl object constructor
        Ssimpl simpl = new Ssimpl(32, "LITTLE_ENDIAN", 512, 512);

        // SIMPL communications and processing here
    }
}
```

**int nameAttach(String name) *Jsimpl only***

**int nameAttach(int port, String host, String name) *Ssimpl only***

*Description:* The nameAttach() method is required by every process that expects to send/receive messages to/from any other process. It must precede any other SIMPL library calls excepting the constructor. The string 'name' is the desired SIMPL name of the process. This name must be unique on each host but not necessarily on a network. The Ssimpl version is used strictly with Windows hosts in mind that need to communicate with other non-Windows hosts; this is the reason for the need to enter the port number used in conjunction with the tclSurrogate and the name of the host computer where the tclSurrogate resides. It returns 0 on success or -1 on failure.

*Example One:* The calling process wants to Jsimpl attach the SIMPL name "RECEIVER".

```
class JavaReceiver
{
    public static void main( String arg [] )
    {
        int ret;

        // default constructor
        Jsimpl simpl = new Jsimpl();

        /*
        perform a SIMPL name attach which is always necessary
        prior to any SIMPL methods being used
        */
        ret = simpl.nameAttach("RECEIVER");
        if (ret == -1)
```

```

    {
        System.out.println("name attach error");
        System.exit(-1);
    }

    // SIMPL communications and processing here
}
}

```

*Example Two:* The calling process wants to Ssimpl attach the SIMPL name "SENDER" using port 50000 and SIMPL sending to the local host via the loop-back interface. This would be a good construction for testing purposes. In this case, the sending, receiving and tclSurrogate programs would all be running in the same host.

```

class JavaSender
{
    public static void main( String arg[] )
    {
        int ret;

        // default constructor
        Ssimpl simpl = new Ssimpl();

        ret = simpl.nameAttach(50000, "localhost", "SENDER");
        if (ret == -1)
        {
            System.out.println("name attach error");
            System.exit(-1);
        }

        // SIMPL communications and processing here
    }
}

```

**int nameDetach()**

*Description:* The nameDetach() method releases all SIMPL functionality from a program that has previously run a successful nameAttach(). If a program is finishing or SIMPL capability is no longer needed, this method should be called prior to termination. Normally, this method is called internally upon program exit. It returns 0 on success or -1 on failure.

*Example:* The following process has successfully name attached SIMPL name "SENDER" and either wants to detach the SIMPL functionality or it is terminating.

```
class JavaSender
{
    public static void main( String arg [] )
    {
        int ret;

        // default constructor
        Jsimpl simpl = new Jsimpl();

        ret = simpl.nameAttach("SENDER");
        if (ret == -1)
        {
            err = simpl.whatsMyError();
            System.out.println("name attach error: " +err);
            System.exit(-1);
        }

        // SIMPL communications and processing here

        ret = simpl.nameDetach();
    }
}
```

<b>int surDetach(int id) <i>Jimpl only</i></b>
--

*Description:* The surDetach() method must be called if a process exits after running a successful nameLocate() call for another process on a remote host. The surDetach() will cause the surrogates to terminate by sending them SUR\_CLOSE messages. The number of receiver surrogates is stored in a table internal to the sending process. Under normal circumstances this will be called internally by the nameDetach() method in the case of process termination. The id passed into the method is the surrogate receiver FIFO id. This method is called as part of the atexit() function within the JNI library for any process that performs a non-local nameLocate(). It returns 0 for success and -1 for failure.

*Example:* The process name locates another process called "RECEIVER" on remote host 192.168.1.42 via the RS-232 serial protocol. Later, it releases the surrogates thus breaking the connection made by the name locate call.

```
class JavaSender
{
    public static void main( String arg [] )
    {
        int id;

        Jsimpl simpl = new Jsimpl();

        // name attach
        if ( simpl.nameAttach("SENDER") == -1)
        {
            System.out.println("name attach error");
            System.exit(-1);
        }

        id = simpl.nameLocate(" 192.168.1.42:RECEIVER" );
        if (id == -1)
        {
            System.out.println("name locate error");
            System.exit(-1);
        }

        // SIMPL communications and processing here

        simpl.surDetach(id);
        simpl.nameDetach();
    }
}
```

<b>int nameLocate(String name)</b>
------------------------------------

*Description:* The nameLocate() method must be called prior to a Send() call so that the sender may connect to the receiver it wishes to send to. The string "name" is defined as follows:

1. Jsimpl: name = protocolName:hostname:processName which contains up to three colon delimited parts where the protocolName determines the network protocol used. If this is not set then a default is chosen. The default will be the first available protocol found in the protocol router's table. The hostName is the name of the host that the sender wishes to send to. It may be either a real host name (canonical or alias) or non-existent in the case of a local send. It can also be in the dotted network

format. If the local host name/IP is used then SIMPL will run locally. The processName is the name of the receiving process that is name attached on that particular host.

2. Ssimpl: name = processName. The process name is the name of the remote process that will be part of the ensuing SIMPL communications.

It returns the id of the receiver process that a message is to be sent to  $\geq 0$  or -1 for failure.

*Example One:* This is an example of where the name of the process "RECEIVER" is running on the local host.

```
class JavaSender
{
    public static void main( String arg [] )
    {
        int ret;
        int receiver;

        // default constructor
        Jsimpl simpl = new Jsimpl();

        ret = simpl.nameAttach("SENDER");
        if (ret == -1)
        {
            System.out.println("name attach error");
            System.exit(-1);
        }

        receiver = simpl.nameLocate("RECEIVER");
        if (receiver == -1)
        {
            System.out.println("name locate error");
            System.exit(-1);
        }

        // SIMPL communications and processing here

        ret = simpl.nameDetach();
    }
}
```

*Example Two:* This is an example of where the process of interest is name attached as "RECEIVER", running on a remote host with IP address 192.168.1.1, and the RS-232 protocol is desired.

```

class JavaSender
{
    public static void main(String arg[])
    {
        int ret;
        int receiver;

        // default constructor
        Jsimpl simpl = new Jsimpl();

        ret = simpl.nameAttach("SENDER");
        if (ret == -1)
        {
            System.out.println("name attach error");
            System.exit(-1);
        }

        receiver = simpl.nameLocate("SIMPL_RS232:192.168.1.1:RECEIVER");
        if (receiver == -1)
        {
            System.out.println("name locate error");
            System.exit(-1);
        }

        // SIMPL communications and processing here

        ret = simpl.nameDetach();
    }
}

```

*Example Three:* This program called JavaSender uses a Ssimpl instantiation to remotely communicate with another networked host with IP address 192.1.68.1.2 via port 50000. It SIMPL name attaches as "SENDER" via the tclSurrogate and name locates the remote program on host 192.168.1.2 with SIMPL name "RECEIVER".

```

class JavaSender
{
    public static void main(String arg[])
    {
        int ret;
        int receiver;

        // default constructor
        Ssimpl simpl = new Ssimpl();

```

```

ret = simpl.nameAttach(50000, "192.168.1.2", "SENDER");
if (ret == -1)
{
    System.out.println("name attach error");
    System.exit(-1);
}

receiver = simpl.nameLocate("RECEIVER");
if (receiver == -1)
{
    System.out.println("name locate error");
    System.exit(-1);
}

// SIMPL communications and processing here

ret = simpl.nameDetach();
}
}

```

### **int Receive()**

*Description:* The Receive() method receives messages from senders in a memory buffer that is allocated during instantiation. This buffer is 1024 bytes by default but can be changed via the overloaded constructor. The contents of this buffer are read by using the available unpacking methods. This method returns a value of  $\geq 0$  for the sender identification or  $\leq -2$  which indicates a proxy has been received or -1 for a failure. In the case of a proxy being sent you must call the method returnProxy(int ret) to obtain the true value of the proxy where ret is the Receive() return value. Note that at the time of printing, proxies are only supported by Jsimpl and not by Ssimpl.

*Example One:* This process is receiving a message composed of one integer, unpacks the integer, packs a reply message being the square of the integer, and replies back to the sender.

```

class JavaReceiver
{
    public static void main(String arg[])
    {
        int ret;
        int sender;
    }
}

```

```
// default constructor
Simpl simpl = new Simpl();

// perform a SIMPL name attach
ret = simpl.nameAttach(50000, "192.168.1.1", "RECEIVER");
if (ret == -1)
{
    System.out.println("name attach error");
    System.exit(-1);
}

while (true)
{
    // receive any incoming SIMPL messages
    sender = simpl.Receive();
    if (sender == -1)
    {
        System.out.println("receive error");
        System.exit(-1);
    }

    // unpack the incoming message from a sender
    int number = simpl.unpackInt(simpl.SIMPL_BIN);

    // pack the outgoing reply to the sender
    simpl.packInt(number * number, simpl.SIMPL_BIN);

    // reply to the sender
    ret = simpl.Reply(sender);
    if (ret == -1)
    {
        System.out.println("reply error");
        System.exit(-1);
    }

    // realign internal input/output buffers for next round
    simpl.reset();
}
}
```

*Example Two:* This process is receiving a proxy. *Jimpl only.*

```
class JavaReceiver
{
    public static void main( String arg [] )
    {
        int ret;
        int sender;
        int proxy;

        // default constructor
        Jimpl simpl = new Jimpl();

        // required so that senders can find this program
        ret = simpl.nameAttach("RECEIVER");
        if (ret == -1)
        {
            System.out.println("name attach error");
            System.exit(-1);
        }

        while (true)
        {
            // receive any incoming SIMPL messages
            sender = simpl.Receive();
            if (sender == -1)
            {
                System.out.println("receive error");
                System.exit(-1);
            }
            else if (sender <= -2)
            {
                proxy = returnProxy(sender);
            }

            // no Reply() required on proxy

            // realign internal input/output buffers for next round
            simpl.reset();
        }
    }
}
```

int Send(int id)
------------------

*Description:* The Send() method is a blocked send. This means that the calling program expects a reply from the receiver and waits until it gets one. The id is the result of a prior call to nameLocate() with regard to the receiver of interest. The message that is sent is composed with the pack() methods and contained within an internal buffer. This buffer defaults to 1024 bytes but can vbe changed to suit by using the overloaded constructor. This method returns the size of the reply message  $\geq 0$  in bytes for success and -1 for a failure.

*Example:* This process is sending a message to another local process called "RECEIVER" and expects a reply. See also Example One for Receive().

```
class JavaSender
{
    public static void main( String arg [] )
    {
        int ret;
        int receiver;

        // default constructor
        Jsimpl simpl = new Jsimpl();

        ret = simpl.nameAttach("SENDER");
        if (ret == -1)
        {
            System.out.println("name attach error");
            System.exit(-1);
        }

        receiver = simpl.nameLocate("RECEIVER");
        if (receiver == -1)
        {
            System.out.println("name locate error");
            System.exit(-1);
        }

        // construct message
        int number = 2;
        simpl.packInt(number, simpl.SIMPL_BIN);

        ret = simpl.Send( receiver );
        if (ret == -1)
        {
            System.out.println("send error");
        }
    }
}
```

```

        System.exit(-1);
    }

    // deconstruct reply (4 hopefully)
    int square = simpl.unpackInt(simpl.SIMPL_BIN);

    // quit
    ret = simpl.nameDetach();
}
}

```

### **int Reply(int sender)**

*Description:* The Reply() method responds a return message from a receiver to a blocked sender. The reply message is contained within an internal buffer and is constructed with the use of the pack() methods. The method returns 0 for success and -1 for failure.

*Example:* See Example One for Receive().

### **int Trigger(int id, int proxy) *Jsimpl only***

*Description:* The Trigger() method sends a proxy identified as an int to a receiving process identified by a prior call to name\_locate() giving id. It is important that the proxy *MUST* be an integer > 0 AND < 7FFFFFFF. This can be thought of as a kick to the receiver and requires no reply. This method returns 0 on success and -1 for a failure.

*Example:* This process is triggering another local process called "RECEIVER" with proxy value of 37.

```

class JavaSender
{
    public static void main(String arg[])
    {
        int ret;
        int receiver;

        // default constructor
        Jsimpl simpl = new Jsimpl();

        ret = simpl.nameAttach("SENDER");
    }
}

```

```
if (ret == -1)
{
    System.out.println("name attach error");
    System.exit(-1);
}

receiver = simpl.nameLocate("RECEIVER");
if (receiver == -1)
{
    System.out.println("name locate error");
    System.exit(-1);
}

ret = simpl.Trigger(receiver, 37);
if (ret == -1)
{
    System.out.println("trigger error");
    System.exit(-1);
}

// quit
ret = simpl.nameDetach();
}
```

**int returnProxy(int proxyNumber)**

*Description:* The returnProxy() method returns the true value of a proxy just received by a Receive(). The proxyNumber is the value  $\leq -2$  returned by the instance of the Receive() call. The proxy number returned will be  $\geq 1$ . This curious arrangement arises due to the fact that the return value of the Receive() method is  $\geq 0$  for messages, null or otherwise. Failures are indicated by -1. This leaves only values  $\leq -1$  if we want to keep Receive() as a simple integer method. Accordingly, when a sender uses Trigger() to send a proxy with a value 7 say (recall that proxies must be an integer 0), the Trigger() method sends it to the receiver as -7. The Receive() method will then return a value of -8. Plugging this value into returnProxy() returns 7, the value of the original proxy.

*Example:* See Example Two in Receive().

**String whatsMyError() *Jimpl only***

*Description:* The whatsMyError() method returns a descriptive string based on the last value of \_simpl\_errno; similar to strerror(errno) in ‘C’.

*Example:* See the example in nameDetach() for a failed nameAttach().

**int getMsgSize(int senderID)**

*Description:* The getMsgSize() method returns the size of a received message based on the senderID; useful for checking.

### F.2.1 Main Method Summary Table

A table of all of the currently available Java SIMPL methods follows for handy reference.

Jimpl Methods	Simpl Methods
int nameAttach(String)	int nameAttach(int, String, String)
int nameDetach()	int nameDetach()
int surDetach(int)	n/a
int nameLocate(String)	int nameLocate(String)
int Send(int)	int Send(int)
int Receive()	int Receive()
int Reply(int)	int Reply(int)
int Trigger(int, int)	n/a
int returnProxy(int)	int returnProxy(int)
String whatsMyError()	n/a
int getMsgSize(int)	int getMsgSize(int)

Table F.2: Summary of Jimpl/Simpl Methods

## F.3 Packing/Unpacking Method Synopsis

Sending and receiving messages via SIMPL can be thought of in terms of:

1. Actions which construct and deconstruct the messages.
2. Actions which transfer these messages to and fro.

The packing/unpacking methods pertain to 1.

Java contains eight primitive data types along with matching arrays which can be used to form the content of data messages. Given the design nature of Java, it is natural to construct and deconstruct the messages via methods based upon these data primitives. The two Java classes, Jsimpl and Ssimpl both contain these methods and are the only way to construct and deconstruct messages.

Errors only occur when using these methods if attempts are made to pack/unpack data outside the confines of the memory buffers. Always be certain that the internal memory buffers have enough memory to accommodate the largest messages expected.

### F.3.1 Packing Methods

The packing methods all take the form *packType(val, offset)* for single primitives and *packTypeArray(value[], offset)* for arrays. The type is variable primitive such as byte, short, float etc. Val is the actual value of the primitive type. The offset defines the sort of packing involved. If offset = SIMPL\_BIN then a binary packing scheme is used. If offset = SIMPL\_CHR then the datum is loaded into the message at the first available memory position. If the offset is  $\geq 0$  then the datum is placed at that position within the outgoing message. All offsets are with respect to the start of the memory buffer which corresponds to the starting point of the message. A method returning 0/-1 indicates success/failure respectively.

### F.3.2 Unpacking Methods

The unpacking methods all take the forms *val = unpackType(offset)* for primitives and *unpackTypeArray(value[], offset)* for arrays. The type is the variable primitive. Val is the actual value of the variable. The offset defines the physical position of the value to be retrieved. If offset = SIMPL\_BIN then a binary scheme is assumed. If offset = SIMPL\_CHR then the datum is read from the message at the first available memory position. If the offset is  $\geq 0$  then the datum is read from that position within the incoming message.

The array unpacking methods return 0/-1 indicating success/failure respectively. The byte, short, int, long, float and double unpacking methods return the value if successful and -1 if not successful. If the actual value is -1 then there is no way to know if there has been an error. The string unpack method returns

the string if successful or NULL for failure. The char unpack returns the char value for success or 0 for failure.

### F.3.3 Packing/Unpacking Method Summary Table

Table F.3 summarizes all of the currently available Java SIMPL packing and unpacking.

Packing Methods		
Method	Purpose	Return
int packByte(byte p, offset o)	Put byte p at offset o	0/-1
int packChar(char p, offset o)	Put char p at offset o	0/-1
int packShort(short p, offset o)	Put short p at offset o	0/-1
int packInt(int p, offset o)	Put int p at offset o	0/-1
int packLong(long p, offset o)	Put long p at offset o	0/-1
int packFloat(float p, offset o)	Put float p at offset o	0/-1
int packDouble(double p, offset o)	Put double p at offset o	0/-1
int packString(String p, offset o)	Put String p at offset o	0/-1
int packByteArray(byte[ ] p, offset o)	Put byte array p at offset o	0/-1
int packCharArray(char[ ] p, offset o)	Put char array p at offset o	0/-1
int packShortArray(short[ ] p, offset o)	Put short array p at offset o	0/-1
int packIntArray(int[ ] p, offset o)	Put int array p at offset o	0/-1
int packLongArray(long[ ] p, offset o)	Put long array p at offset o	0/-1
int packFloatArray(float[ ] p, offset o)	Put float array p at offset o	0/-1
int packDoubleArray(double[ ] p, offset o)	Put double array p at offset o	0/-1
Unpacking Methods		
Method	Purpose	Return
byte unpackByte(offset o)	Get byte p at offset o	byte
char unpackChar(offset o)	Get char p at offset o	char
short unpackShort(offset o)	Get short p at offset o	short
int unpackInt(offset o)	Get int p at offset o	int
long unpackLong(offset o)	Get long p at offset o	long
float unpackFloat(offset o)	Get float p at offset o	float
double unpackDouble(offset o)	Get double p at offset o	double
int unpackString(int l, offset o)	Get String length l at offset o	String
int unpackByteArray(byte[ ] p, offset o)	Get byte array p at offset o	0/-1
int unpackCharArray(char[ ] p, offset o)	Get char array p at offset o	0/-1
int unpackShortArray(short[ ] p, offset o)	Get short array p at offset o	0/-1
int unpackIntArray(int[ ] p, offset o)	Get int array p at offset o	0/-1
int unpackLongArray(long[ ] p, offset o)	Get long array p at offset o	0/-1
int unpackFloatArray(float[ ] p, offset o)	Get float array p at offset o	0/-1
int unpackDoubleArray(double[ ] p, offset o)	Get double array p at offset o	0/-1

Table F.3: Java Packing/Unpacking Methods

### F.3.4 Example

For our example a Java receiver and a ‘C’ sender for SIMPL communications has been chosen. It can be assumed that both programs are local to the same host. Given that the sender is a ‘C’ program the natural messaging type is binary based, hence the use of a structure as the message template. Note that the message contains a Java *char* data type. On the ‘C’ sender’s side, there is no such data type and so we concoct one with the aid of a union. The message employs all of the Java primitive data types for both sending and replying in order to provide an example of the usage of all currently available packing/unpacking methods. Since we are SIMPL communicating locally with a ‘C’ sender, the Jsimpl library is the best choice for this Java receiver.

This example is using binary based packing/unpacking to keep consistency with a ‘C’ sender using a binary structure as its message template. However, message could have been text based by setting the packing/unpacking offset fields to SIMPL\_CHR. As well, the offset field could also have been set to a number greater than or equal to zero for a user-defined offset between message data members. In either of these cases however, the message structure on the ‘C’ side would have to be padded where necessary or done away with and replaced by a series of strategically placed *get-type* operations.

#### Java Packing/Unpacking Example

```
1 // a 'C' SIMPL sender messaging with a Java SIMPL receiver
2
3 // system headers
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <simpl.h>
8
9 // function prototypes
10 void buildMsg(void);
11 void printReply(void);
12
13 // needed for java char primitive
14 typedef union
15 {
16     signed char x[2];
17     short y;
18 } JCHAR;
19
20 // the binary message template
```

```
21 typedef struct
22 {
23     char a;
24     JCHAR b;
25     short c;
26     int d;
27     long e;
28     float f;
29     double g;
30     char h[31];
31     char i[3];
32     JCHAR j[3];
33     short k[3];
34     int l[3];
35     long m[3];
36     float n[3];
37     double o[3];
38 } MSG;
39
40 // incoming/outgoing message memory
41 char inMem[1024];
42 char outMem[1024];
43
44 int main()
45 {
46     int receiver;
47     int ret;
48
49 // SIMPL name attach
50 if (name_attach("SENDER", NULL) == -1)
51 {
52     printf("cannot attach name: %s\n", whatsMyError());
53     exit(-1);
54 }
55
56 // name locate the target receiver
57 receiver = name_locate("RECEIVER");
58 if (receiver == -1)
59 {
60     printf("cannot locate receiver: %s\n", whatsMyError());
61     exit(-1);
62 }
63
64 // construct binary message to be sent to "RECEIVER"
65 buildMsg();
66
```

```
67 // send the message to "RECEIVER"
68 ret = Send( receiver , outMem, inMem, sizeof(MSG) , sizeof(MSG));
69 if (ret == -1)
70 {
71     printf("Send error: %s\n" , whatsMyError());
72     exit(-1);
73 }
74
75 // print contents of reply message from "RECEIVER" to screen
76 printReply();
77
78 return(0);
79 }
80
81 void buildMsg()
82 {
83     MSG *ptr = (MSG *)outMem;
84
85     ptr->a = 79;
86     ptr->b.x[0] = 80;
87     ptr->b.x[1] = 0;
88     ptr->c = 81;
89     ptr->d = 82;
90     ptr->e = 83;
91     ptr->f = 84;
92     ptr->g = 85;
93     memcpy(ptr->h, "We are the knights who say nee!", 31);
94     ptr->i[0] = 66;
95     ptr->i[1] = 79;
96     ptr->i[2] = 66;
97     ptr->j[0].x[0] = 65;
98     ptr->j[0].x[1] = 0;
99     ptr->j[1].x[0] = 66;
100    ptr->j[1].x[1] = 0;
101    ptr->j[2].x[0] = 68;
102    ptr->j[2].x[1] = 0;
103    ptr->k[0] = 97;
104    ptr->k[1] = 98;
105    ptr->k[2] = 99;
106    ptr->l[0] = 100;
107    ptr->l[1] = 101;
108    ptr->l[2] = 102;
109    ptr->m[0] = 103;
110    ptr->m[1] = 104;
111    ptr->m[2] = 105;
112    ptr->n[0] = 106;
```

```

113 ptr->n[1] = 107;
114 ptr->n[2] = 108;
115 ptr->o[0] = 109;
116 ptr->o[1] = 110;
117 ptr->o[2] = 111;
118 }
119
120 void printReply()
121 {
122 MSG *ptr = (MSG *)inMem;
123
124 printf("\nReplied message ... \n");
125 printf("a=%c\n", ptr->a);
126 printf("b=%c %c\n", ptr->b.x[0], ptr->b.x[1]);
127 printf("c=%d\n", ptr->c);
128 printf("d=%d\n", ptr->d);
129 printf("e=%ld\n", ptr->e);
130 printf("f=%f\n", ptr->f);
131 printf("g=%f\n", ptr->g);
132 printf("h=%31s\n", ptr->h);
133 printf("i=%c %c %c\n", ptr->i[0], ptr->i[1], ptr->i[2]);
134 printf("j=%c%c %c%c %c%c\n", ptr->j[0].x[0], ptr->j[0].x[1],
135     ptr->j[1].x[0], ptr->j[1].x[1], ptr->j[2].x[0],
136     ptr->j[2].x[1]);
137 printf("k=%d, %d, %d\n", ptr->k[0], ptr->k[1], ptr->k[2]);
138 printf("l=%d, %d, %d\n", ptr->l[0], ptr->l[1], ptr->l[2]);
139 printf("m=%ld, %ld, %ld\n", ptr->m[0], ptr->m[1], ptr->m[2]);
140 printf("n=%f,%f,%f\n", ptr->n[0], ptr->n[1], ptr->n[2]);
141 printf("o=%f,%f,%f\n", ptr->o[0], ptr->o[1], ptr->o[2]);
142 fflush(stdout);
143 }
```

lines 4-42 Required headers, function prototypes, global variables etc.

lines 14-18 Create a union to deal with the Java "char" data type which does not exist in 'C'. The Java "byte" data type corresponds to a "signed char" in 'C'.

lines 21-28 A structure template for the message to be sent and replied. This indicates binary type messages.

lines 50-54 SIMPL name attach.

lines 58-62 SIMPL name locate.

line 65 Call a function to build the outgoing message.

lines 68-73 Send the mesage to the Java receiver.

line 76 Print out the reply message contents to stdout.

```
1 // a Java SIMPL receiver messaging with a 'C' SIMPL sender
2
3 class Receiver
4 {
5     public static void main(String arg[])
6     {
7         int ret;
8         int sender;
9
10        // default constructor
11        Jsimpl simpl = new Jsimpl();
12
13        /*
14         * perform a SIMPL name attach which is always necessary
15         * prior to any SIMPL methods being used
16         */
17        ret = simpl.nameAttach("RECEIVER");
18        if (ret == -1)
19        {
20            System.out.println("name attach error");
21            System.exit(-1);
22        }
23
24        while (true)
25        {
26            // receive any incoming SIMPL messages
27            sender = simpl.Receive();
28            if (sender == -1)
29            {
30                System.out.println("receive error");
31                System.exit(-1);
32            }
33
34            // unpack the incoming message from the sender
35            unpackInput(simpl);
36
37            // build the outgoing reply to the sender
38            packOutput(simpl);
39}
```

```
40 // reply to the sender
41 ret = simpl.Reply(sender);
42 if (ret == -1)
43 {
44     System.out.println("reply error");
45     System.exit(-1);
46 }
47 }
48 }

49

50 // a method for extracting a known SIMPL message
51 static void unpackInput(Jsimpl simpl)
52 {
53     // unpack the incoming message
54     byte A = simpl.unpackByte(simpl.SIMPL_BIN);
55     char B = simpl.unpackChar(simpl.SIMPL_BIN);
56     short C = simpl.unpackShort(simpl.SIMPL_BIN);
57     int D = simpl.unpackInt(simpl.SIMPL_BIN);
58     long E = simpl.unpackLong(simpl.SIMPL_BIN);
59     float F = simpl.unpackFloat(simpl.SIMPL_BIN);
60     double G = simpl.unpackDouble(simpl.SIMPL_BIN);
61     String H = simpl.unpackString(31, simpl.SIMPL_BIN);
62     byte[] I = new byte[3];
63     simpl.unpackByteArray(I, simpl.SIMPL_BIN);
64     char[] J = new char[3];
65     simpl.unpackCharArray(J, simpl.SIMPL_BIN);
66     short[] K = new short[3];
67     simpl.unpackShortArray(K, simpl.SIMPL_BIN);
68     int[] L = new int[3];
69     simpl.unpackIntArray(L, simpl.SIMPL_BIN);
70     long[] M = new long[3];
71     simpl.unpackLongArray(M, simpl.SIMPL_BIN);
72     float[] N = new float[3];
73     simpl.unpackFloatArray(N, simpl.SIMPL_BIN);
74     double[] O = new double[3];
75     simpl.unpackDoubleArray(O, simpl.SIMPL_BIN);

76

77 // print out the contents of the incoming message
78 System.out.println("Received message ...");
79 System.out.println("A=" +A);
80 System.out.println("B=" +B);
81 System.out.println("C=" +C);
82 System.out.println("D=" +D);
83 System.out.println("E=" +E);
84 System.out.println("F=" +F);
85 System.out.println("G=" +G);
```

```
86     System.out.println("H=" +H);
87     System.out.println("I=" +I[0]+ " "+I[1]+ " "+I[2]);
88     System.out.println("J=" +J[0]+ " "+J[1]+ " "+J[2]);
89     System.out.println("K=" +K[0]+ " "+K[1]+ " "+K[2]);
90     System.out.println("L=" +L[0]+ " "+L[1]+ " "+L[2]);
91     System.out.println("M=" +M[0]+ " "+M[1]+ " "+M[2]);
92     System.out.println("N=" +N[0]+ " "+N[1]+ " "+N[2]);
93     System.out.println("O=" +O[0]+ " "+O[1]+ " "+O[2]);
94 }
95
96 // a method for building the replied response
97 static void packOutput(Jsimplesimpl simpl)
98 {
99     byte a = 80;
100    simpl.packByte(a, simpl.SIMPL_BIN);
101    char b = 66;
102    simpl.packChar(b, simpl.SIMPL_BIN);
103    short c = 1;
104    simpl.packShort(c, simpl.SIMPL_BIN);
105    int d = 9;
106    simpl.packInt(d, simpl.SIMPL_BIN);
107    long e = 6;
108    simpl.packLong(e, simpl.SIMPL_BIN);
109    float f = 11F;
110    simpl.packFloat(f, simpl.SIMPL_BIN);
111    double g = 4;
112    simpl.packDouble(g, simpl.SIMPL_BIN);
113    String h = "We are the knights who say noo!";
114    simpl.packString(h, simpl.SIMPL_BIN);
115    byte[] i = {97, 98, 99};
116    simpl.packByteArray(i, simpl.SIMPL_BIN);
117    char[] j = {50, 51, 52};
118    simpl.packCharArray(j, simpl.SIMPL_BIN);
119    short[] k = {50, 51, 52};
120    simpl.packShortArray(k, simpl.SIMPL_BIN);
121    int[] l = {201, 301, 401};
122    simpl.packIntArray(l, simpl.SIMPL_BIN);
123    long[] m = {-201, -301, -401};
124    simpl.packLongArray(m, simpl.SIMPL_BIN);
125    float[] n = {201F, 301F, 401F};
126    simpl.packFloatArray(n, simpl.SIMPL_BIN);
127    double[] o = {-201, -301, -401};
128    simpl.packDoubleArray(o, simpl.SIMPL_BIN);
129 }
130 }
```

line 11 Create an instance of Jsimpl.

lines 17-22 SIMPL name attach.

lines 24-51 Loop around receiving and unpacking messages, packing and replying responses.

lines 27-32 Receive messages.

line 35 Decompose the received message print the contents out to screen.

line 38 Compose the reply message.

lines 41-46 Reply response back to sender.

## F.4 Miscellaneous Methods

There are six so-called miscellaneous methods. The first two are the class instance creators viz. `Jsimpl()` and `Ssimpl()`. These are necessary for any SIMPL communications as they set up the correct instance. They are both overloaded methods. Examples of their use can be seen in the preceding examples as their inclusion is mandatory for every Java program that uses SIMPL.

The next two miscellaneous methods are concerned with the underlying memory buffers which contain the messages that are being packed/unpacked. The methods `getInBufferSize()` and `getOutBufferSize()` return the total amount of memory available for incoming and outgoing messages respectively. Recall that these values are set to 1024 bytes by default in the case of the constructors within `Jsimpl()` and `Ssimpl()` or to a user-defined value in the case of the overloaded constructors. These methods would be most useful when ascertaining what the default values are in the case of future changes to the underlying functions.

Lastly, the method `getOutSizeAvailable()` returns the current amount message memory still available when packing a message. This is useful for ensuring that a memory overrun does not occur when building outgoing messages.

### F.4.1 Miscellaneous Method Summary Table

Table F.4 summarizes all of the currently available miscellaneous Java SIMPL methods.

Miscellaneous Methods		
Method	Purpose	Return
Jimpl()	Jimpl instance constructor	Jimpl
Jimpl(int i, int o)	Jimpl instance constructor	Jimpl
Simpl()	Simpl instance constructor	Simpl
Simpl(int a, String b, int i, int o)	Simpl instance constructor	Simpl
int getInBufferSize()	Get max. incoming memory size	int $\geq 0$
int getOutBufferSize()	Get max. outgoing memory size	int $\geq 0$
int getOutSizeAvailable()	Get available outgoing memory	int $\geq 0$

Table F.4: Java Miscellaneous Methods



# Appendix G

## Tcl Library Functions

### G.1 Procedure Synopsis

The following contain descriptions of the Tcl SIMPL library calls offered at the time of printing.

#### **name\_attach processName**

*Description:* The name\_attach function is required by every process that expects to send/receive messages to/from any other process. It is responsible for setting up all of the SIMPL functionality that may be required. It must precede any other Tcl/Tk SIMPL library calls. The string processName is the desired SIMPL name of the process. This name must be unique on each host but not necessarily on a network. A process that has successfully called this function is often referred to as being 'name attached'. The length of processName is defined in the base 'C' SIMPL library.

The return code from name\_attach depends on the Tcl/Tk package used. If the 'C' shared library (fctclx.so) is used then the return code will be a string containing, SIMPL\_ID.process\_ID. The shared library version of name\_attach returns a NULL string if the underlying SIMPL name\_attach fails.

If the fcgateway library package is used the return code will be the pid field in the name\_attach protocol message. As long as the tclSurrogate daemon process is running this name\_attach call should succeed.

*Example:* The calling process wants to attach the SIMPL name "SENDER"

using the fctclx shared library.

```
#!/usr/bin/wish

lappend auto_path $env(SIMPL_HOME)/lib
package require Fctclx

set myName SENDER

set myslot [name_attach $myName]
```

*Example:* The calling process wants to attach the SIMPL name "SENDER" using the tclSurrogate libraries.

```
#!/usr/bin/wish

set gatewayAddr 127.0.0.1
set gatewayPort 8000

lappend auto_path $env(SIMPL_HOME)/lib
package require fcgateway
package require fcsocket

set myName SENDER

set mypid [name_attach $myName]
```

### **name\_detach**

*Description:* The name\_detach function removes SIMPL related components from a program that has previously run a successful name\_attach. This call should be made prior to exiting from the program. If the fctclx shared library is used this call will be made automatically by the SIMPL library. If the fcgateway/tclSurrogate library is used care must be taken to place the name\_detach in the exit path.

*Example:* The calling process wants to detach the SIMPL name "SENDER" using the fctclx shared library.

```
#!/usr/bin/wish
```

```
lappend auto_path $env(SIMPL_HOME)/lib
package require Fctclx

set myName SENDER

set myslot [name_attach $myName]

... processing

name_detach
exit
```

*Example:* The calling process wants to detach the SIMPL name "SENDER" using the tclSurrogate libraries.

```
#!/usr/bin/wish

set gatewayAddr 127.0.0.1
set gatewayPort 8000

lappend auto_path $env(SIMPL_HOME)/lib
package require fcgateway
package require fcsocket

set myName SENDER

set mypid [name_attach $myName]

... processing

name_detach
exit
```

### **name\_locate processName**

*Description:* The name\_locate function must be called prior to a Send or Trigger call to open the SIMPL communications channel to the intended receiver. If the fctclx shared library is used, then the SIMPL composite name string *name:protocolName:hostName:processName* is permitted as the argument. If the fcgateway library is used the straight processName is the only permitted form for the argument. The name\_locate returns with the ID of the receiver process. A failure will return a 0 or -1 as the ID. (see 'C' library reference)

*Example:* This is an example of where the name of the process of interest is running on the local host.

```
...
set recvName BOBR
set recvID [name_locate $recvName]
...
```

### Receive

*Description:* The Receive function receives messages from senders. If the fctclx shared library is used the message buffer size is fixed to 2k and is maintained internal to the shared library. The fcgateway library has a 64k maximum message size as the message field in the protocol is a two byte integer. In both cases the Receive function will return a binary string containing: fromID numberbytes message. Failures are indicated by a -1 as the number of bytes.

*Example:* This is an example of a typical Receive call.

```
...
set buf [Receive]
binary scan $buf i1i1 fromWhom nbytes
binary scan $buf x8a$nbytes msg
... process msg
... build reply message (rMsg) of length rBytes
Reply $fromWhom $rMsg $rBytes
```

### Send id outbuf outsize

*Description:* The Send function is a blocked send. This means that the calling program expects a reply from the receiver and waits until it gets one. The id is the result of a prior call to name\_locate with regard to the receiver of interest. The outgoing message is contained in outbuf. The outgoing message size is

outsize. The response to Send will be lead by an 8 byte header which can be interpreted as two integers. The first of these will be the internal slot used by the shared library for this message and the second will be the number of bytes contained in the response. In the event of an error the slot field will be set to -1.

*Example:* This is a typical Send call.

```
...
set rMsg [Send $targetID $sMsg $sBytes]
binary scan $rMsg i1i1a* slot rbytes rmsg
...
```

### Reply fromwhom replybuf replysize

*Description:* The Reply function responds a return message from a receiver to a blocked sender indicated by the fromwhom channel. The reply message is contained in the replybuf and the size is replysize. Since it is difficult to intercept errors on TCP/IP sockets (the gateway version uses sockets) the Reply call will not return any error information.

*Example:* A typical Reply sequence.

```
set rMsg [binary format "s1a*" \
$MYTOKEN(REGISTER) \
$mypixel]

set rBytes [string length $rMsg]

Reply $fromWhom $rMsg $rBytes]
...
```

### Trigger id proxy

*Description:* The Trigger function is currently only supported in the Tcl/Tk shared library. Trigger sends a proxy identified as an int to a receiving process

identified by a prior call to name\_locate() giving id. It is important that the proxy *MUST* be an integer > 0 AND < 7FFFFFFF. This can be thought of as a kick to the receiver and requires no reply. This function returns 0 on success and -1 for a failure. Failures can occur due to:

1. The calling process was never name attached,
2. Receiver does not exist,
3. Fifo errors, and
4. Proxy number is out of range.

*Example:* This process is triggering another local process called 'receiver'.

```
...
Trigger $targetID $myproxy
...
```

### **logit traceloggerID file function mask logmask msg**

*Description:* The logit call interfaces into the SIMPL fclogger trace logger. This trace logger is denoted by the channel ID returned from a previous name\_locate call. The file and function fields allow for identification as to where the trace log message originated. The mask bits interact with the logmask to determine if the message will be transmitted. The actual trace log message is contained in the msg field.

*Example:* A trace log call.

```
...
set logMask 0xff
set TRACEMASK(MISC) 0x10
set this simplbook
...
set fn toggleX10

logit $loggerID $this $fn $TRACEMASK(MISC) $logMask \
  [format "current temp=%d C limit=%d C" $mytemp $templimit]

...
```

## G.2 Procedure Summary Table

A table of all of the currently available Tcl SIMPL library procedures is included for handy reference.

Procedure	Purpose	Return
name_attach name	Start SIMPL functionality	binary string
name_detach	End SIMPL functionality	n/a
name_locate string	Make a SIMPL connection	0/-1
Receive	Receive a SIMPL message	binary string
Send receiverId buf bufsize	Send a SIMPL message	binary string
Reply senderId buf bufsize	Reply a SIMPL message	n/a
Trigger receiverId proxy	Send a proxy	0/-1
logit traceLoggerID file fn mask lmask msg	Send log msg to traceLogger	n/a

Table G.1: Summary of Tcl Procedures



# Appendix H

## TCP/IP Surrogate - Details

In this appendix we address the operational details of the TCP/IP surrogate.

### H.1 Operation

Suppose that we have the situation that a sender process exists on Host 1 and a receiver process exists on Host 2. The sender needs to send messages to the receiver and will use a TCP/IP connection. Let us examine the operation of the surrogates by following the numbers in Figure H.1.

1. The sender performs a ***name\_locate***("SIMPL-TCP:Host 2:receiver") call in order to make a connection to the remote receiver. During this function call a message is sent to the protocolRouter program inquiring as to whether a TCP/IP surrogate receiver (*surrogate\_r*) is available for use. If so, the SIMPL name of this TCP/IP surrogate receiver is replied back to the sender.
2. *surrogate.R* then forks another TCP/IP *surrogate\_r* in readiness for any forthcoming requests.
3. The sender ***name\_locate*** call, now knowing the SIMPL name of its prospective surrogate receiver (*surrogate\_r*) then sends a remote name locate message to *surrogate\_r*.
4. *surrogate.r* then opens a TCP/IP socket to *surrogate\_S* on Host 2. *surrogate\_r* then sends a name locate message to *surrogate\_S*. Upon receiving this message *surrogate\_S* forks a *surrogate\_s* child which completes the surrogate socket pair, viz. *surrogate\_r/surrogate\_s*. Upon a successful fork,

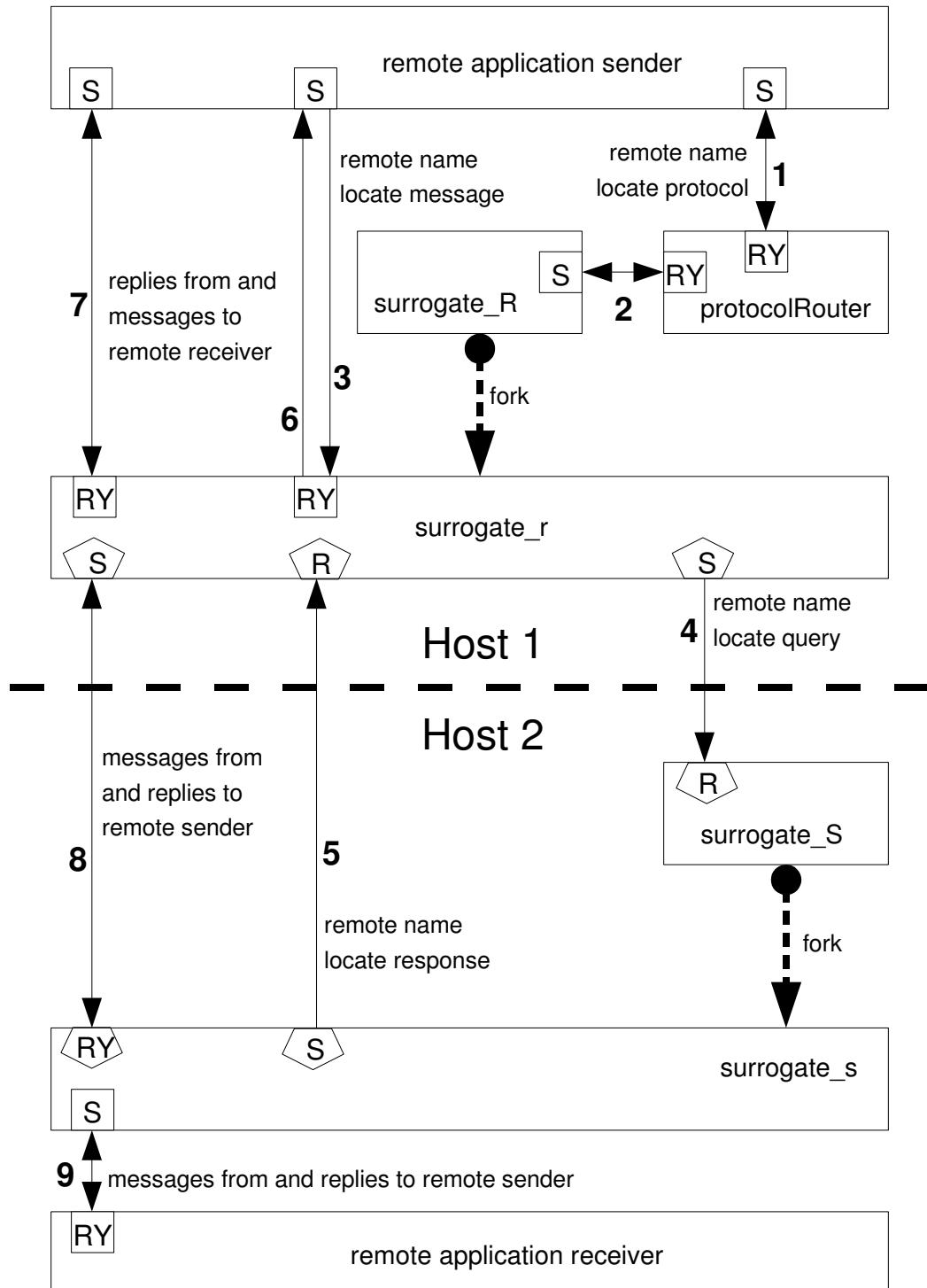


Figure H.1: TCP/IP Surrogate Networking

surrogate\_s then has the original name locate message of its parent surrogate\_S. surrogate\_s then performs a local name locate on the receiver process.

5. surrogate\_s replies back to the surrogate\_r the result of the name locate.
6. surrogate\_r replies back to the result of the remote name locate for the receiver process. **IMPORTANT** Steps 1-6 above are all contained within the original **name\_locate**("Host 2:SIMPL\_TCP:receiver") call made by the sender. This is what *lies under the hood*.
7. The sender sends messages to and receives replies from the surrogate receiver, surrogate\_r. As far as the sender is concerned, it is the remote receiver that it is in contact with.
8. The surrogate receiver, surrogate\_r and the surrogate sender surrogate\_s, exchange messages and replies.
9. The surrogate sender, surrogate\_s, sends messages to the receiver program and receives replies in return. As far as the receiver program is concerned it is in direct contact with the sender.

For a complete description of the tokenized messages internal to the surrogates see Appendix K.

## H.2 Starting Up

The first thing that is required when running surrogate programs is the presence of the protocol router program. For details on the protocol router, see Section 7.3. It is started and run in the background as follows: **protocolRouter &**

The next thing is to start the TCP/IP surrogate program: surrogateTcp. It can be started and run in the background as follows: **surrogateTcp &.** This program has a number of command line arguments defined as follows:

- a The A port used to bind sockets to; defaults to 8001.
- b The B port used to bind sockets to; defaults to A port.
- i The width of an integer, 4 bytes for a 32-bit integer, 8 bytes for a 64-bit integer; defaults to 4 bytes.

- k The value of the keep alive time out in seconds; defaults to 10. A value of 0 indicates no keep alive is to be run.
- n The value of a name locate time out in seconds; defaults to 60. This is the allowable time taken for a remote name locate call before timing out and returning a failure.

*Example One:* surrogateTcp -k0 -n120 &

This example turns off the keep alive mechanism and increases the name locate time out to two minutes on what is perhaps a slow network.

*Example Two:* surrogateTcp -a8020 -k60 &

This example changes the port to 8020, perhaps due to a port confliction with some other device and slows down the keep alive to once per minute.

# Appendix I

## RS-232 Surrogate - Details

In this appendix we address the operational details of the RS-232 surrogate.

### I.1 Operation

Suppose again as in the Appendix H on TCP/IP surrogates that we have the situation that a sender process exists on Host 1 and a receiver process exists on Host 2. The sender needs to send messages to the receiver via an RS-232 connection. Let us examine the operation of the surrogates by following the numbers in Figure I.1.

1. The sender performs a ***name\_locate***("Host2:SIMPL\_RS232:receiver") call in order to make a connection to the remote receiver. During this call a message is sent to the protocolRouter program inquiring as to whether a TCP/IP surrogate receiver (*surrogate\_r*) is available for use. If so, the SIMPL name of this TCP/IP surrogate receiver is replied back to the sender.
2. *surrogate\_R* then forks another TCP/IP surrogate\_r in readiness for any forthcoming requests.
3. The sender ***name\_locate*** call, now knowing the SIMPL name of its prospective surrogate receiver (*surrogate\_r*) then sends a remote name locate message to *surrogate\_r*.
- 4a. *surrogate\_r* then sends the remote name locate message to the program called *rs232\_rw*. This program reads from and writes to the serial port.

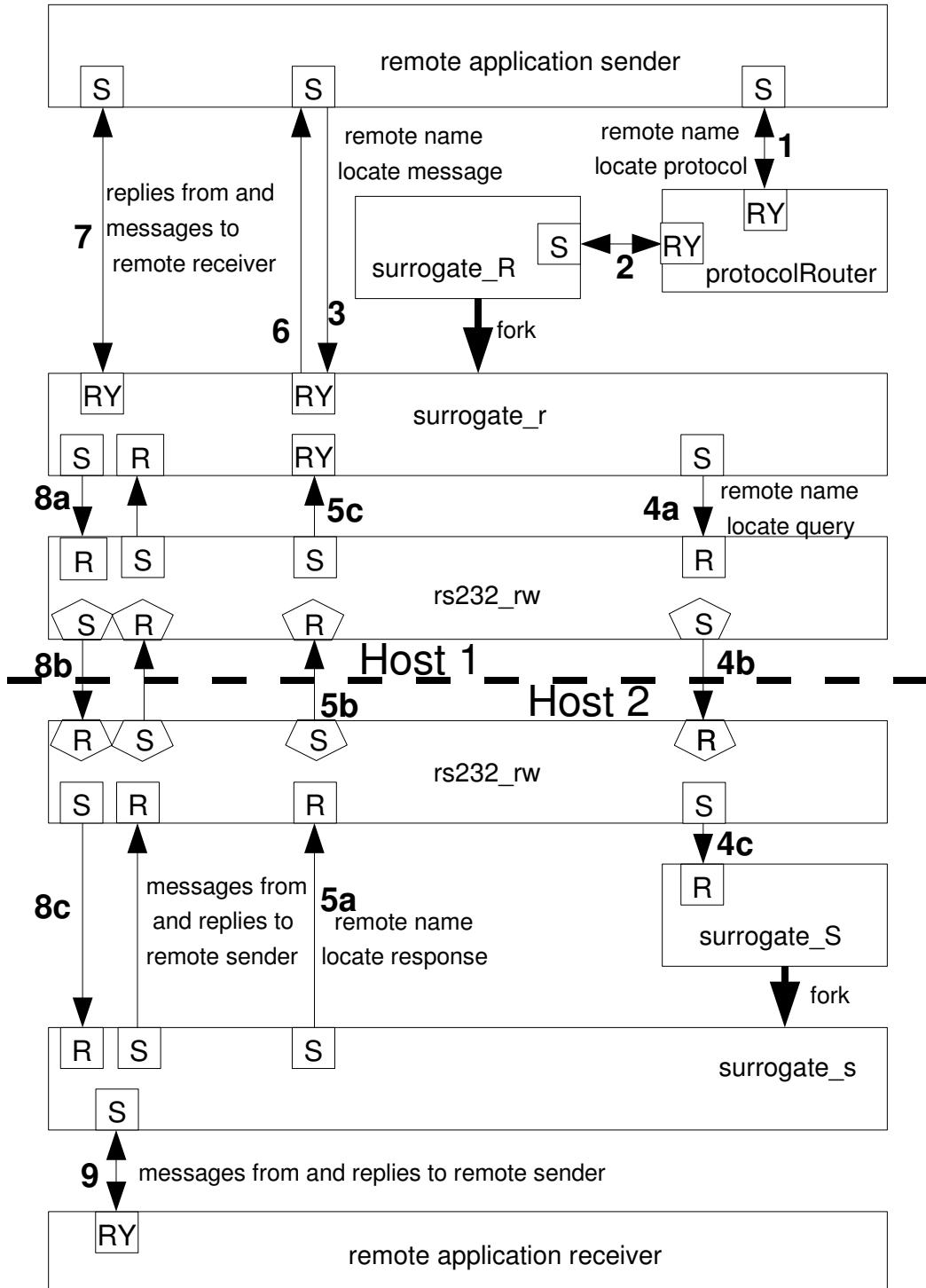


Figure I.1: RS-232 Surrogate Networking

- 4b. rs232\_rw then writes the remote name locate message out the serial port where the message is then read by its counterpart on Host 2.
- 4c. rs232\_rw having received the remote name locate message then sends the message on to surrogate\_S. Upon receiving this message surrogate\_S forks a surrogate\_s child which completes the surrogate socket pair, viz. surrogate\_r/surrogate\_s. Upon a successful fork, surrogate\_s then has the original name locate message of its parent surrogate\_S. surrogate\_s then performs a local name locate on the receiver process.
- 5a. surrogate\_s replies the results of the name locate back to rs232\_rw.
- 5b. rs232\_rw then writes this result out the serial line to its counterpart on Host 1.
- 5c. Upon receiving the result of the name locate message from rs232\_rw on Host 2, rs232\_rw on Host1 then forwards the message on to surrogate\_r.
6. surrogate\_r replies back the result of the remote name locate for the remote receiver process. **IMPORTANT:** Steps 1-6 above are all contained within the original name\_locate("Host 2:SIMPL\_RS232:receiver") call made by the sender. This is *what lies under the hood*.
7. The sender sends messages to and receives replies from the surrogate receiver, surrogate\_r. As far as the sender is concerned, it is the remote receiver that it is in contact with.
8. The surrogate receiver, surrogate\_r, rs232\_rw on Host 1, and rs232\_rw on Host 2 and the surrogate sender surrogate\_s exchange messages and replies.
9. The surrogate sender surrogate\_s, sends messages to the receiver program and receives replies in return. As far as the receiver program is concerned it is in direct contact with the sender.

## I.2 Starting Up

As in the section on the TCP/IP surrogates, the first thing that is required when running surrogate programs is the presence of the protocol router program. For details on the protocol router, see Section 7.3. It is started and run in the background as follows: **protocolRouter &**.

Furthermore, the rs232\_rw program must also be started. It is started and run in the background as follows: **rs232\_rw &**. This program has a number of command line arguments defined as follows:

**-i**: the width of an integer, 4 bytes for a 32-bit integer, 8 bytes for a 64-bit integer; defaults to 4 bytes.

**-s**: the name of the serial device to be used. The default value is /dev/ttyS0.

**-v**: the boolean value of the verbosity. It defaults to 0 which is off. The verbosity prints to stdout the sorts of messages that are being read off or written to the serial port.

**IMPORTANT:** Note that it is NOT the responsibility of SIMPL to set the appropriate serial port values. There are a number of ways and utility programs to do this depending on the operating system. The default port opening values etc. can be found in a file called rs232.h in the SIMPL development tree.

*Example:* rs232\_rw -s/dev/ttyS1 -v &

This example sets the serial port to /dev/ttyS1 and turns on the verbosity.

The next thing is to start the RS-232 surrogate program: surrogateRS232. It can be started and run in the background as follows: **surrogateRS232 &**. This program has a number of command line arguments defined as follows:

**-i**: the width of an integer, 4 bytes for a 32-bit integer, 8 bytes for a 64-bit integer; defaults to 4 bytes.

**-k**: the value of the keep alive time out in seconds; defaults to 10. A value of 0 indicates no keep alive.

**-n**: the value of a name locate time out in seconds; defaults to 60. This is the allowable time taken for a remote name locate call before timing out and returning a failure.

*Example:* surrogateRS232 -k0 -n120 &

This example turns off the keep alive mechanism and increases the name locate time out to two minutes on what is perhaps a slow network.

# Appendix J

## tclSurrogate Protocol

The so-called tclSurrogate protocol is so named because it was first used to connect a Tcl applet to the SIMPL framework. Since that time the embedded protocol has been used in many other instances than Tcl applets (Tcl/Tk Windows -> SIMPL, VB Windows -> SIMPL, IO Anywhere network appliance -> SIMPL).

The concept of the tclSurrogate is really very straightforward. The tclSurrogate parent is started up and listens for connections on a predefined TCP/IP port. (default to port 8000). When a connection is made and accepted the tclSurrogate forks a child process whose purpose is to act as the SIMPL interface to the rest of the SIMPL application. The process making the connection and this child communicate with each other over the TCP/IP socket which exists between them. The structures they use to communicate on this socket is what we are terming the tclSurrogate protocol.

### J.1 General Message Format

The basic protocol message format is as follows:

token(2)	nbytes(2)	ID(4)	data(token dependant)
----------	-----------	-------	-----------------------

where the number in brackets represents the size of the field in bytes.

## J.2 Some Terminology

We also need to decide on some naming conventions for the various elements involved in a message pass via the tclSurrogate child process.

There are basically three players that are involved:

- socket connected app (**sockapp**)
- tclSurrogate child (**child**)
- SIMPL sender or receiver (**SIMPL sender or receiver**)

When illustrating the protocol in detail below we will be using these short hand forms for these players.

## J.3 Tokens

The following tokens form the basis of this protocol. The master source for this information is in the ‘C’ header located at `$SIMPL_HOME/tcl/include/surroMsgs.h`

Token	Value
NAME_ATTACH	0
NAME_DETACH	1
NAME_LOCATE	2
SEND	3
REPLY	4
RELAY	5
IS_LOGGER_UP	6
LOGIT	7
SEND_NO_REPLY	8
ACK	9
PING	10

## J.4 NAME\_ATTACH

sockapp -> child

0	32	n/a(4)	SIMPL name(20)	n/a(4)	n/a(4)
---	----	--------	----------------	--------	--------

child -> sockapp

4	32	-1	SIMPL name(20)	pid(4)	slot(4)
---	----	----	----------------	--------	---------

When the sockapp issues the NAME\_ATTACH message it has the effect of setting the SIMPL name on the child process.

## J.5 NAME\_DETACH

sockapp -> child

1	0
---	---

child -> sockapp

4	4	-1
---	---	----

The NAME\_DETACH causes the child process to exit after a 2 second delay.

## J.6 NAME\_LOCATE

sockapp -> child

2	28	n/a(4)	SIMPL name(20)	n/a(4)
---	----	--------	----------------	--------

child -> sockapp

4	28	-1	SIMPL name(20)	rc(4)
---	----	----	----------------	-------

The NAME\_LOCATE will have the effect of performing a local *name\_locate* for the SIMPL name. The rc field will contain the result of that call.

## J.7 SEND

sockapp -> child

3	4+sbytes	toWhom(4)	sdata(sbytes)
---	----------	-----------	---------------

child -> SIMPL receiver(toWhom)

sdata(sbytes)
---------------

SIMPL receiver -> child

rdata(rbytes)
---------------

child -> sockapp

4	4+rbytes	-1	rdata(rbytes)
---	----------	----	---------------

## J.8 REPLY

The use of this token has been detailed in the other paragraphs.

## J.9 RELAY

SIMPL sender(fromWhom) -> child

sdata(sbytes)
---------------

child -> sockapp

5	4+sbytes	fromWhom(4)	sdata(sbytes)
---	----------	-------------	---------------

sockapp -> child

4	4+rbytes	fromWhom(4)	rdata(rbytes)
---	----------	-------------	---------------

child -> SIMPL sender(fromWhom)

rdata(rbytes)
---------------

## J.10 IS\_LOGGER\_UP

sockapp -> child

6	28	n/a(4)	trace logger name(20)	n/a(4)
---	----	--------	-----------------------	--------

child -> sockapp

4	28	-1	trace logger name(20)	loggerID(4)
---	----	----	-----------------------	-------------

NOTE: the trace logger name isn't currently checked in the child code, so this call will return the global loggerID variable for the tclSurrogate parent's trace logger irrespective of the supplied trace logger name.

## J.11 LOGIT

sockapp -> child

7	52+msglen	loggerID(4)	fileName(20)	funcName20)	mask(4)	globalMask(4)
---	-----------	-------------	--------------	-------------	---------	---------------

There is no response to the LOGIT call.

## J.12 SEND\_NO\_REPLY

sockapp -> child

8	4+sbytes	toWhom(4)	sdata(sbytes)
---	----------	-----------	---------------

child -> SIMPL receiver(toWhom)

sdata(sbytes)
---------------

The **Reply** from the SIMPL receiver is simply discarded in this instance.

## J.13 ACK

sockapp -> child

9
---

The ACK is not responded to.

## J.14 PING

sockapp -> child

10
----

child -> sockapp

4	4	-1
---	---	----



# Appendix K

## Surrogate Internal Messages

SIMPL surrogates communicate with each other via tokenized message passing. See Chapter 5 for a detailed description of tokenized message passing. Table K.1 contains a list of the various tokens that are used.

The following is a list of the more important tokens, what they mean and where they are used.

**SUR\_NAME\_LOCATE** This token starts out in a remote name locate call. It then goes to the relevant surrogate\_r program. From there it goes to the surrogate\_S program on the remote host. This causes surrogate\_S to fork a surrogate\_s program which then carries out a local name locate for the desired receiver process.

**SUR\_SEND** This token originates in surrogate\_r. By setting this token surrogate\_r is telling its surrogate\_s partner that a sender-originated message is following.

**SUR\_REPLY** This is the message token sent from surrogate\_s to its partner surrogate\_r implying that the following message is a reply to the previously sent message.

**SUR\_CLOSE** If a surrogate\_r gets a proxy to shutdown from its local sender or if surrogate\_r detects that the sender has disappeared then it sends a message with this token set to its surrogate\_s partner. If the sender has terminated, then neither of the surrogates are required and this tokenized message tells surrogate\_s to exit.

**SUR\_PROXY** This token originates in surrogate\_r. By setting this token surrogate\_r is telling its surrogate\_s partner that a sender originated proxy is

Token	Value	Purpose
SUR_NAME_ATTACH	0	deprecated
SUR_NAME_DETACH	1	deprecated
SUR_NAME_LOCATE	2	name locate requests
SUR_SEND	3	send message
SUR_REPLY	4	reply message
SUR_CLOSE	5	surrogate quit
SUR_PROXY	6	proxy message
SUR_ERROR	7	error message
SUR_ALIVE	8	keep alive message
SUR_ALIVE_REPLY	9	keep alive reply message
SUR_SURROGATE_READY	10	message to protocol router
SUR_REQUEST_PROTOCOL	11	message to protocol router
SUR_DUMP_TABLE	12	message to protocol router

Table K.1: Surrogate Tokens

following.

**SUR\_ERROR** This token originates in surrogate\_s. By setting this token surrogate\_s is telling its surrogate\_r partner that some sort of problem has occurred.

**SUR\_ALIVE** Both surrogate\_r and surrogate\_s send this message token to each other as a keep alive inquiry.

**SUR\_ALIVE\_REPLY** Both surrogate\_r and surrogate\_s send this message token to each other as a response to a keep alive inquiry.

**SUR\_SURROGATE\_READY** This token represents a message from surrogate\_R to the protocol router stating that a new surrogate\_r is ready for use.

**SUR\_REQUEST\_PROTOCOL** This message type originates in a remote name locate call whereby a message is sent to the protocol router asking for the SIMPL name of an available surrogate\_r of the desired protocol.

**SUR\_DUMP\_TABLE** This message type is sent by the utility program called *dumpProtocolTable* to the protocol router. This message causes the protocol router to reply the contents of its internal surrogate\_r table which the *dumpProtocolTable* program displays to stdout.

# Appendix L

## Apache Configuration

In order for the Apache web server to work correctly with our CGI Sudoku example it may be necessary for the Apache configuration file to be modified. On a Linux system, the configuration file is called *httpd.conf*. Note that there may be several or many other configuration files used by Apache as well. This file may be found in /etc/httpd/conf in the case of a general installation or /opt/lampp/etc in the case of a default LAMPP installation or elsewhere. In the following, we are going to use a default LAMPP installation as a basis for the directory names; in fact, there is a directive in the configuration file called *ServerRoot*. For our examples this root directory will be called */opt/lampp*. In fact, it could really be anything as long as Apache knows about it.

The basic Apache configuration file *httpd.conf* is to be found in */opt/lampp/etc* along with others.

The *html* files are to be found in */opt/lampp/htdocs*.

The following directives should be in the Apache configuration file:

1. This line should appear in the configuration file so that the CGI dynamic shared object (DSO) can be used by Apache:

**LoadModule cgi\_module modules/mod\_cgi.so**

2. A useful shortcut for Apache is to define what is meant by the standard directory listing for cgi-bin, the location of the CGI programs. This also appears as a line in the configuration file:

**ScriptAlias /cgi-bin/” /opt/lampp/cgi-bin/”**

3. You will need to be able to access the CGI programs in this directory so you will need the following as well:

```
<Directory "/opt/lampp/cgi-bin">
    AllowOverride None
    Options None
    Require all granted
</Directory>
```

4. In order to access SIMPL libraries it may be necessary to export a shell variable to Apache. Specifically, if the CGI program is a python script then the python interpreter will need to know where the SIMPL modules are located. This is often accomplished using the shell variable PYTHONPATH. For example, at boot time the following shell command may be run from a profile:

**export PYTHONPATH=/home/simpl/python/modules**

If this is the case, then the following line may also be required in the Apache configuration file:

**PassEnv PYTHONPATH**

It has been found that for some installations that the PassEnv directive does not function well; in that case, use:

**SetEnv PYTHONPATH /home/simpl/python/modules**

5. All SIMPL libraries require a special directory whose location is given by an environment variable called FIFO\_PATH. This variable must also be set, usually at boot time and must also be passed to Apache. Suppose that **FIFO\_PATH=/home/fifo**, then use:

**PassEnv FIFO\_PATH or SetEnv FIFO\_PATH=/home/fifo**

6. It may also be necessary to add the following line in the case of a python-based CGI script if your CGI script name ends in .py:

**AddHandler cgi-script .py**

# Appendix M

## Trace Logger

Tracking down anomalies in executing code is a subject for a book on its own. Source code debuggers have their place as do strategically placed temporary printf()*s*. However, both are highly intrusive to the executable itself.

The source code debuggers require that the code be compiled in a specific way. The compiler adds special code into the executable to allow that executable to be put under the control of another program: the debugger. This allows the programmer to set break points, step through code, examine variables etc.

For quick checking of program flow or variable contents, strategically placed printf()*s* are often used. However, this requires that the executable be modified at the source level and then recompiled before testing. Most often those printf()*s* need to be removed before the executable undergoes final testing and is deployed.

Neither of these debugging techniques is useful for doing any trouble shooting on a deployed executable already in the field. For this we need a trace logger. The ideal trace logger would:

- Minimally impact the executable performance thereby allowing trace logger calls to be liberally sprinkled throughout production code.
- Allow for verbosity of the trace log output to be altered on a running executable.
- Allow for multiple levels of information to be individually enabled or disabled.

The SIMPL core toolkit comes with a very capable trace logger which for historical reasons is called **fclogger**.

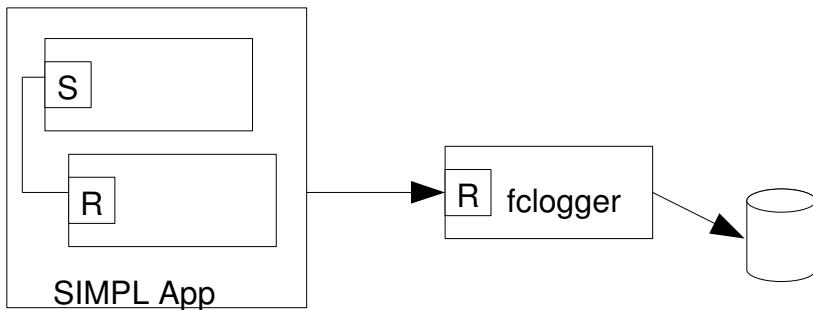


Figure M.1: Trace Logger

This trace logger comes in two parts:

- A very basic SIMPL receiver which receives a text message and redirects that to the standard output.
- A simpllog API which *hides* the sending half behind a *printf-like* API call.

```
fcLogx(char *filename,
char *function,
unsigned int globalMask,
unsigned int userMask,
char *format,
... )
```

where

**filename** - user supplied name of file (often just `__FILE__`)

**function** - user supplied name of local function

**globalMask** - 32 bit mask which allows for 32 separate logging "levels" to be enabled/disabled across the application

**userMask** - 32 bit mask indicating which mask level(s) apply to this particular call

**format** - full printf style format

**...** - full printf style variable list

The idea is that you substitute the fcLogx() call in your code wherever you would normally have put a printf for debugging and/or error display. The userMask bit is set on each fcLogx() call by the programmer when the source code is written. The globalMask bits are set by any combination of:

- default global variable
- command line override
- special SIMPL message for mask override

Only messages where the globalMask and userMask bits are both set are transmitted to the trace logger. Otherwise the fcLogx() simply returns immediately after the mask comparison.<sup>1</sup>

The **fclogger** is a very basic SIMPL receiver. When it receives a message it simply dumps the entire contents of that message out to the console screen (stdout) and sends back a NULL Reply.

Of course it is a simple matter of redirecting that output to get it to appear inside a file. In chapter 9 we show an example of the trace logger output being redirected to a file.

This SIMPL trace logger is a very powerful tool. Through its use one can provide very powerful feedback about the inner workings of the code in ways that are much less intrusive than a source code debugger. Furthermore, you can leave your trace debugging statements in your code without incurring an appreciable performance penalty.

In chapter 9 we illustrated the creation of test0001. Watch what happens when we make a very basic change to our test0001 script to add another command line argument to the stimulator invocation line.

```
stimulator -n BOBS -r BOBR -m 0x0 -l LOGGER -b &
```

The addition of the -m argument has stopped the flow of log messages to the trace log file. We didn't have to do any recompilation to get this!

With the tokenized SIMPL messaging into the receiver it would be a simple exercise to design a message which would allow you to alter the trace log mask while the receiver was running. In other words, we could suppress or enable the flow of log messages on the fly without stopping and restarting the SIMPL modules.

---

<sup>1</sup>For examples of the SIMPL trace logger in action you can consult any of the softwareICs which are fully fclogger enabled.



# Appendix N

## Advanced SIMPL for C Programmers

A complete Send/Receive/Reply transaction in SIMPL involves four `memcpy()` operations:

1. outgoing buffer into shared memory in sender,
2. shared memory into incoming buffer in receiver,
3. reply buffer into shared memory in receiver,
4. shared memory into incoming buffer in sender.

In most cases this is the preferred mode of operation for a SIMPL program. Messages are organized in such a fashion that accidental message overwrite is less likely. However, there are legitimate cases whereby it is advantageous to manipulate the SIMPL message directly in the shared memory thereby eliminating the computational cost of extra buffers and `memcpy()` calls. One example is a SIMPL architecture with a receiver gateway/router. In that case it makes sense for the gateway receiver to simply *peek* into the message and then relay it on to the destination based on some information such as a message token. Another example would be a packet streaming case with larger SIMPL message/packet sizes. In this case the computational cost for `memcpy()` calls and extra buffer memory may be significant.

Since manipulating the SIMPL shared memory directly involves pointers, it is restricted to SIMPL C programmers. Furthermore since the same shared memory area is used by both the outgoing message and its response, this type

of programming requires the utmost care and therefore is recommended only for advanced C programmers.

## N.1 Background

Each of the following three function calls involved in a SIMPL transaction,

- **int Send(int id, void \*out, void \*in, unsigned outSize, unsigned inSize)**
- **int Receive(char \*\*ptr, void \*inArea, unsigned maxBytes)**
- **int Reply(char \*ptr, void \*outArea, unsigned size)**

have void \* buffers defined. In each case if the programmer supplies a valid pointer as these arguments, the SIMPL core library performs the appropriate memcpy() to or from the SIMPL shared memory automatically. If instead of a pointer the NULL argument is used (and a non-zero message size is matched), this is a signal to the core library that the programmer will be manipulating the shared memory buffer directly.

In the next section the example code from the core chapter is modified to manipulate the SIMPL message entirely in shared memory.

## N.2 Sample Code

In Chapter 4 a very basic example of a sender-receiver pair was illustrated. This example will now be modified to illustrate the manipulation of the messages entirely in shared memory. Firstly, the C sender:

### The ‘C’ Sender

```

1 // 'C' sender: program called c-sender
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <simpl.h>
7
8 int main()
9 {
10    int receiver;
```

```
11 int *inNumber;
12 int i;
13 int *outNumber;
14 int size = sizeof(int); // S1
15 char *me = "SENDER";
16
17 // perform simpl name attach
18 if (name_attach(me, NULL) == -1) // S2
19 {
20     printf("%s: cannot attach name=%s\n", me, whatsMyError());
21     exit(-1);
22 }
23
24 // name locate the receiver program
25 receiver = name_locate("RECEIVER"); // S3
26 if (receiver == -1)
27 {
28     printf("%s: cannot locate receiver=%s\n", me, whatsMyError());
29     exit(-1);
30 }
31
32 // sender creates the shared memory first time it is used hence
33 // we have to Send once before we can use the shared memory pointer
34 i=1;
35 if (Send(receiver, &i, NULL, size, size) == -1) // S5
36 {
37     printf("%s: cannot send to receiver=%s\n", me, whatsMyError());
38     exit(-1);
39 }
40
41 outNumber = (int *)whatsMyShmPtr();
42 inNumber = (int *)whatsMyShmPtr();
43
44 // build message and send to receiver
45 for (i = 1; i <= 10; i++) // S4
46 {
47     *outNumber = i;
48
49     if (Send(receiver, NULL, NULL, size, size) == -1)// S5
50     {
51         printf("%s: cannot send to receiver=%s\n", me, whatsMyError());
52         exit(-1);
53     }
54
55     printf("out number=%d in number=%d\n", i, *inNumber); // S6
56 }
```

```

57
58 name_detach ();
59 return (1); // S7
60 }

```

There are at least two *gotcha's* in this piece of code. First of all, the shared memory buffer is created by a Send call upon its first use. In this example there is no other Send happening ahead of the main loop (eg. a trace logger call) so an explicit first Send must be taken outside the for loop. That Send cannot utilize the NULL argument to indicate that shared memory is being manipulated for the outgoing message because that shared memory hasn't yet been created. Subsequent calls to Send can utilize this NULL argument indicating that the message is being manipulated directly in the shared memory. Secondly, by virtue of the printf() inside the loop wanting to display both the outNumber and inNumber an extra local variable had to be defined to store the outNumber to avoid this being trumped when the shared memory buffer area is reused to house the inNumber. These are classic illustrations as to why such direct manipulation in the shared memory is not for the *faint of heart*. The modified receiver is below:

### The ‘C’ Receiver

```

1 // 'C' receiver: program called c_receiver
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <simpl.h>
7
8 int main()
9 {
10     char *sender;
11     int *inNumber;
12     int *outNumber;
13     int size = sizeof(int); // R1
14     char *me = "RECEIVER";
15
16     // perform simpl name attach
17     if (name_attach(me, NULL) == -1) // R2
18     {
19         printf("%s: cannot attach name=%s\n", me, whatsMyError());
20         exit(-1);
21     }
22
23     while (1)
24     {

```

```

25 // receive incoming messages
26 if (Receive(&sender, NULL, size) == -1) // R3
27 {
28     printf("%s: Receive error-%s\n", me, whatsMyError());
29     continue;
30 }
31 inNumber = (int *)whatsThisShmPtr(sender);
32 outNumber = inNumber; // same shared memory
33
34 // calculate square of sent number
35 *outNumber = *inNumber * (*inNumber); // R4
36
37 // reply squared number to sender
38 if (Reply(sender, NULL, size) == -1) // R5
39 {
40     printf("%s: Reply error-%s\n", me, whatsMyError());
41     continue;
42 }
43 }
44
45 name_detach(); // R6
46 return(1);
47 }
```

Once again the outNumber and inNumber share the same shared memory space. Care must be taken to not assume that the incoming message will persist beyond the Reply. Furthermore the incoming message will get overwritten by that Reply so care must be taken to insure that the message processing algorithm is aware of this fact. Once again not for the *faint of heart*.

These examples represent the extreme end of the *process directly in shared memory* spectrum. Any of the four NULL arguments could be used individually to partially manipulate the message in shared memory. One of the most common such partial manipulations would occur in a message router such as the example below:

### The ‘C’ Router

```

1 // ‘C’ router: program called c-receiver
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <simpl.h>
7
8 int main()
```

```

9  {
10 int receiver;
11 char *sender;
12 int *inNumber;
13 int size = sizeof(int); // R1
14 char *me = "ROUTER";
15
16 // perform simpl name attach
17 if (name_attach(me, NULL) == -1) // R2
18 {
19     printf("%s: cannot attach name-%s\n", me, whatsMyError());
20     exit(-1);
21 }
22
23 // name locate the receiver program
24 receiver = name_locate("RECEIVER"); // S3
25 if (receiver == -1)
26 {
27     printf("%s: cannot locate receiver-%s\n", me, whatsMyError());
28     exit(-1);
29 }
30
31 while (1)
32 {
33     // receive incoming messages
34     if (Receive(&sender, NULL, size) == -1) // R3
35     {
36         printf("%s: Receive error-%s\n", me, whatsMyError());
37         continue;
38     }
39     inNumber = (int *)whatsThisShmPtr(sender);
40
41     if (*inNumber < 9)
42         Relay(sender, receiver);
43     else
44     {
45         int outNumber = -1;
46
47         Reply(sender, &outNumber, size); // return error
48     }
49 }
50
51 name_detach(); // R6
52 return(1);
53 }
```

# Appendix O

## Warning and Error Messages

Code	Description
1	No SIMPL name provided for name attach.
2	No fifo path environment variable defined.
3	SIMPL name is probably in use.
4	Unable to create shmem for message communication.
5	Unable to delete shmem from message communication.
6	Unable to attach shmem created for message communication.
7	Unable to detach shmem created for message communication.
8	Unable to remove shmem created for message communication.
9	No SIMPL name has been attached to this process.
10	Error in creating trigger fifo.
11	Error in getting trigger fifo.
12	Error in opening trigger fifo.
13	Error in reading from trigger fifo.
14	Error in writing to trigger fifo.
15	Bad fifo file descriptor.
16	Unable to find the surrogate process.
17	Unable to ascertain current host name.
18	Unable to locate remote process.
19	Received message exceeds receiver buffer allocation.
20	Reply message exceeds sender buffer allocation.
21	Receive/Reply problem ... could be a failed receiver.
22	Requested name too short.
23	Too many colons in requested name locate.

*continued on next page*

Code	Description
24	No system host name set.
25	System host name is too long.
26	Protocol not in router table.
27	No more room in remote receiver table.
28	Unable to open fifo directory.
29	Command line args parsing error.
30	Proxy value must be $\geq 1$ .
31	Local host has no available IP information.

Table O.1: Warning and Error Messages

# Appendix P

## SIMPL License

The SIMPL project essentially uses two licenses:

- the Lesser General Public License (LGPL)
- the Public Domain License

The details of these licenses are available for your perusal in the \$SIMPL\_HOME/license subdirectory.

We want as many people to use SIMPL as possible. We don't want to restrict its usage in commercial custom software in any way. However, we expect SIMPL users to respect the licenses under which different parts of the SIMPL toolset were released.

There is a legal angle to these licenses and we'll defer those debates to better qualified individuals. It is more important that SIMPL users understand the spirit of these licenses as we interpret them.

The SIMPL core library is for example, licensed under the LGPL. The LGPL license essentially says that since we own the copyright to the SIMPL core code we can dictate the terms of usage. The LGPL gives SIMPL core code users generous terms of usage which include full and unfettered access to the source code. The only stipulation that the LGPL insists upon is that if you choose to make changes to the SIMPL core and redistribute those, then you cannot change the terms of the access that we originally gave to you. When stated in this way this seems like a fair bargain.

In practical terms this means that if any user (including commercial custom software vendors) makes changes to the SIMPL core and then redistributes those changes with their application, they cannot restrict access to those changes. We would hope that in those instances the users would do the honourable thing

and recontribute their changes back to the SIMPL project without having to be asked. These terms however, do not extend to the user SIMPL application itself. The code to that portion of a custom application can remain proprietary. We don't make any distinction if the SIMPL application is statically or dynamically linked to the SIMPL core libraries.

The Public Domain sections which include the SIMPL softwareICs, were given freely to users without any conditions. We would encourage users who improve upon the code under this license to contribute back to the SIMPL project but there is no legal obligation to do so.

# Appendix Q

## SRY Kernel Module

### Q.1 Introduction

Earlier in this book, Section 2.1 contained a brief history of SIMPL that mentioned an approach called SIPC. SIPC was an acronym for **S**ynchronized **I**nter**Process Communication. It was an early attempt to speed up message passing by writing a kernel module that took care of much of the mechanics behind this form of communication. It was abandoned for a number of reasons and SIMPL grew out of a different approach where all of the code was based in *user space*. That took place around the year 2000. Since that time much has happened in the kernel development area of Linux programming resulting in a renewed interest in this approach once again.**

Rather than calling the more recent version by the same name, we decided to call it simply SRY for **S**end **R**eceive **r**epl**Y**. Given the length of time between SIPC and SRY it might be guessed that the two don't have too much in common ... but this would be wrong! For all of the changes that have been made in coding practice, the algorithms themselves are essentially the same. So it seems that *what you do* doesn't always change much, only *how you do it* seems to change. This of course assumes that the *what* of your design worked reasonably well in the first place and that the *how* were changes for the better. We think that Linux has come a long way and with a vast number of improvements, some of which are reflected in SRY.

The purpose of this appendix is to:

1. Explain and describe SRY generally.
2. Use SRY in a sender/receiver example.

3. Look at SRY as an example of how to make a kernel module. (After all, the aim of this book is to learn some programming.)

## Q.2 Background

As mentioned earlier, the SIPC core software which partially ran in kernel space was dropped in favour of core software that ran in user space context alone. Recall from Chapter 4 that the core elements are essentially Send, Receive and Reply. We didn't want to have to support two entirely different groups of core software and the Linux kernel module approach was changing rapidly with each new kernel minor version (even versioning has changed). As well, the design of the kernel modules was such that pieces like the *file operations structure* had to be coded in a very particular way and only later became more flexible to changes. Given that we were interested in making software that would run readily on different O/S platforms, kernel based modules are decidedly not the way to do this. Changes within user space libraries are often just a question of recompiling your code for a particular situation as long as the function APIs remain the same.<sup>1</sup> As such, the SIMPL project has been running with very little change since Linux version 2.0.36, other flavours of UNIX and MAC OS X.

When we refer to *kernel space* and *user space* contexts, we are saying that kernel modules run in kernel memory space whereas applications run in user memory space. These two spaces may be thought of as separate entities with their own particular address space. In fact, when data is transferred from user space to kernel space for example, one of the checks found in some of the kernel code is with respect to the veracity of this data's memory address within user space.

SRY is essentially a more modern version of SIPC. As such, SRY is in part a kernel module which can be thought of as a *device driver* for byte streams (messages). This SRY kernel module reacts to system calls made by applications in user space via the user space SRY library. For example, suppose an application program written in 'C' needs to be a SRY *receiver*. The first thing that this program must do is to start its own SRY capabilities. The program will make an *openSRY(name)* function call via the SRY user context library. The *openSRY()* will then proceed to perform some initializing and then make a system call to the SRY module within the kernel that finishes the name attaching procedure.

One way that Linux classifies device drivers is by several broad types such as *character*, *block* and *network*. This sort of classification relies on the type

---

<sup>1</sup>It is also very helpful if coding standards such as ANSI are followed.

of data transference involved. That is, character-type devices transmit streams of bytes, block-type devices transmit 512/1024/2048/etc. blocks of bytes and network-type devices stream packets of bytes, usually between separate host computers.

Given that SIMPL messages can be anything from one byte to almost anything, the SRY module has been chosen to be a character device. The SRY kernel module is accessed via a file system device node called */dev/sry*. That is, a character special file called *sry* exists in the */dev* directory. A long list (*ls -l*) printout of */dev/sry* looks like the following:

```
crw-rw-rw- 1 root root 249, 0 May 7 20:46 sry
```

- The leading **c** indicates that it is a character device.
- **rw-rw-rw-** are the file permissions.
- **1** is the number of links.
- **root** is the owner.
- **root** is the group.
- **249** is the major number.
- **0** is the minor number.
- **May 7 20:46** is the date and time the module was inserted.
- **sry** is the name of the character device file.

## Q.3 SRY Organization

Figure Q.1 shows the relationships between the pieces that make up SRY. Application programs invoke SRY by calling the required functionality available from the SRY user space library. These SRY library functions, such as *Receive()*, make system calls to the SRY kernel module. Generally speaking, *system calls* are the interface between application programs and the kernel.

Let's examine the *User Space Applications*, *User Space SRY Library* and *Kernel Space SRY Module* from Figure Q.1 separately and explore how each connects to make the full picture.

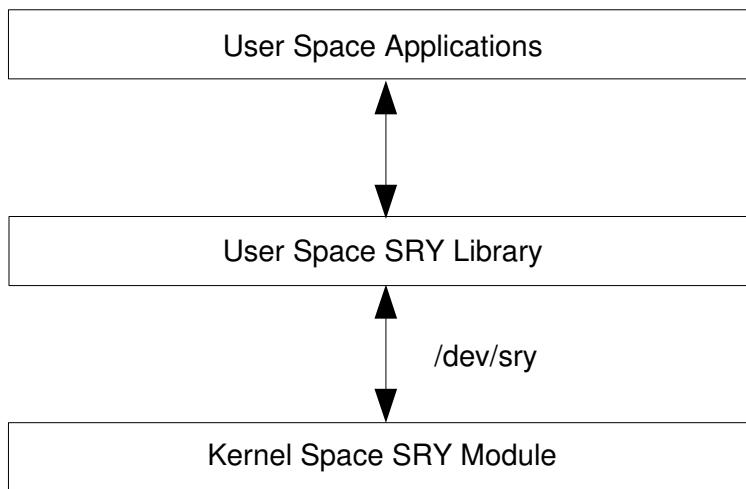


Figure Q.1: SRY Breakdown

### Q.3.1 User Space Applications

User space applications are simply the programs that you write to accomplish some task. Tables Q.1 and Q.2 provide a list of functions that enable SRY messaging between application programs.

Table Q.1 indicates member functions that are contained within the SRY C++ class. Most of the SRY calls can be made through an object instantiated via the C++ SRY class.

Table Q.2 is made up of strictly ‘C’ functions. All of the class member functions listed in Table Q.1 are coded as *wrappers* of these ‘C’ functions. This allows maintenance of one centralized code base.

The two tables also represent the fact that there are two separate libraries; viz. *libsryc* and *libsryccpp*. The former is for ‘C’ application code and the latter is for C++. The C++ library contains the ‘C’ library and more. The ‘C’ library does not contain any C++ aspects. Both libraries are available for static or dynamic linking.

There are a number of reasons why this approach has been taken with respect to the SRY libraries. It allows the option of a procedural ‘C’ or an object-oriented C++ approach to be taken with respect to application programming. This lends a certain latitude with respect to programming tastes. Another reason is that we found it easier and cleaner to *hand wrap* the ‘C’ functions rather than using SWIG, Boost et al. in conjunction with the C++ library for extending SRY to Python. More on this in section Q.5.11.

SRY Member Function	Purpose
SRY(std::string&)	Constructor: Initialize SRY receiver.
SRY(const char *)	Constructor: Initialize SRY receiver.
SRY(void)	Constructor: Initialize SRY sender.
~SRY(void)	Destructor: Remove SRY capabilities.
int Receive(long *, void *, int)	Receive a proxy or a message.
int Reply(long, void *, int)	Return reply to a sent message.
int ReplyError(long)	Reply canned error to a sender.
int returnProxy(int)	Return a proxy's true value.
long Locate(std::string&, std::string&, int)	Return receiver SRY id.
long Locate(const char *, const char *, int)	Return receiver SRY id.
int Send(long, void *, int, void *, int)	Send a message to a receiver.
int Trigger(long, int)	Send a proxy.
int maxRecvSize(long)	Return receiver's msg buffer size.
int maxRplySize(long)	Return sender's reply msg buffer size.
int chkReceiver(long)	Return receiver's status.
int chkSender(long)	Return sender's status.
int fd(void)	Return file descriptor to /dev/sry.

Table Q.1: C++ SRY Class Member Functions

SRY Function Call	Purpose
int openSRY(const char *)	Initialize SRY capabilities.
void closeSRY(void)	Remove SRY capabilities.
int Receive(long *, void *, int)	Receive a proxy or a message.
int Reply(long, void *, int)	Return reply to a sent message.
int ReplyError(long)	Reply canned error to a sender.
int returnProxy(int)	Return a proxy's true value.
long Locate(const char *, const char *, int)	Return receiver SRY id.
int Send(long, void *, int, void *, int)	Send a message to a receiver.
int Trigger(long, int)	Send a proxy.
int maxRecvSize(long)	Return receiver's msg buffer size.
int maxRplySize(long)	Return sender's reply msg buffer size.
int chkReceiver(long)	Return receiver's status.
int chkSender(long)	Return sender's status.
int rfd(void)	Return file descriptor to /dev/sry.
const char *sryErf(void)	Return SRY error string.
void sryLog(const char *, ...)	Write a message to the SRY log file.

Table Q.2: C SRY Functions

Let's discuss the various entries from both tables.

**SRY(std::string &name)**

**SRY(const char \*name)**

**SRY()**

The three above overloaded constructors call openSRY(). The first uses a C++ string object reference to identify a receiver's SRY name. The second uses a 'C' string to identify a receiver's SRY name. The third takes no parameters indicating that a pure sender is initializing SRY capability.

**int openSRY(const char \*name)**

The openSRY() call is required for any program that is going to make use of SRY. The openSRY() performs a number of necessary preliminary tasks as well as registering the program within a receiver table that is contained in the SRY kernel module. This function takes one argument:

**const char \*name** In the case of a program intending to receive messages, this must be a unique and unchanging character string such as "MY\_RECEIVER". Note that it can be anything within the parameters of a 'C' string and must not be longer than fifty characters.

**const char \*name = “”** If a program intends to be a sender only, then passing a null string is the correct method.

The openSRY() returns 0 for success and -1 for failure. Failures can arise for the following reasons:

- Program was interrupted during initialization.
- SRY name is already in use. Names must be unique.
- No room left in receiver table.
- Program has already name attached; only one to a customer.
- SRY name is too long.

- Couldn't open the device node.
- Some sort of failure in kernel communications.

```
~SRY()
```

This is the C++ SRY class destructor. It calls closeSRY() to remove any SRY capabilities. It applies to all programs that use SRY.

```
void closeSRY(void)
```

The closeSRY() call removes a program's SRY capabilities. This function takes no arguments. It returns nothing. Failures can occur due to:

- Program was interrupted during SRY close down.
- Some sort of failure in kernel communications.

```
int Receive(long *id, void *in, int maxSize)
```

The Receive() call receives messages and/or proxies from senders. It will block program execution until a message is received; this is known as *receive blocking*. The function takes three arguments:

**long \*id** A unique SRY identification with respect to the SRY sender that the receiver receives a message from. This value is needed for any reply back to the sender that may follow.

**void \*in** A pointer to a memory buffer where incoming messages are kept within the receiving program.

**int maxSize** The maximum size in bytes an that incoming message can be. It usually corresponds to the size of the incoming message buffer. It is necessary to make the message buffer at least as large as the largest incoming message expected.

This function returns the size of the incoming message or a value which is less than -1 in the case of a proxy. In the case of a failure, -1 is returned. Failures may occur due to:

- Program was interrupted during Receive().
- Program was not name attached prior to the Receive() call.
- Some sort of failure in kernel communications.

```
int Reply(long id, void *out, int msgSize)
```

The Reply() call replies a return message to a sender. It follows a Receive() call after a message has been received. Its purpose is to reply a message, if any, and to release the sending program which awaits this reply. The sending program will remain *reply blocked* until it gets the reply. The function takes three arguments:

**long id** This value uniquely identifies the sender that the receiver is replying to. This is the value that is obtained from the Receive() that came prior.

**void \*out** This is a pointer to memory containing the outgoing reply message intended for the receiver.

**int msgSize** The size of the outgoing reply message in bytes.

Reply() returns the size of the reply message on success or -1 in the case of a failure. Failures are caused by:

- Program was interrupted during Reply().
- The sender's identifier does not match.
- Some sort of failure in kernel communications.

```
int ReplyError(long id)
```

The ReplyError() call replies an error condition to a sender. Its purpose is to report a problem to a *reply blocked* sender. The function takes one argument:

**long id** This value uniquely identifies the sender that the receiver is replying to.

ReplyError() returns 0 on success or -1 in the case of a failure. It is really just a special case of Reply().

<b>int returnProxy(int value)</b>
-----------------------------------

The returnProxy() call is used to derive the true value of a proxy after a Receive() call has received a trigger. The function takes one argument and it is the return value of the Receive() when triggered. A trigger versus a message is always known by the return value from the Receive(). The Receive() returns the message size  $\geq 0$  after receiving a message, a -1 for some sort of failure and a number  $\leq -2$  for a triggered proxy. Passing this last value to the returnProxy function will have the true value of the proxy returned.

So, this function returns a number  $\geq 1$  which is the true value of the proxy or -1 if the value number is  $> -2$ .

<b>long Locate(std::string &amp;host, std::string &amp;name, int size)</b>
--

This is the C++ version of Locate. It calls Locate() below but uses C++ string objects to identify the host computer and the receiver of interest.

<b>long Locate(const char *host, const char *name, int size)</b>
--

The Locate() call is used by senders to find a name attached receiver by name; hence the reason that name attached receiver's names must be unique. The function call takes three arguments:

**const char \*host** The name of the host computer where the receiver of interest resides.

**const char \*name** The name of the receiver the sender wishes to send a message to.

**int size** The maximum size in bytes any sent message may be.

On success, a unique SRY identification with respect to the SRY receiver that the sender wishes to send either a message or a proxy to is returned, ie. an integer  $> 0$ . In the case of failure, -1 is returned. Failures can occur due to:

- Program was interrupted during locate.

- SRY receiver name was not found in the receiver table.
- Some sort of failure in kernel communications.

**int Send(long id, void \*out, int o, void \*in, int i)**

The `Send()` call sends a message to a receiver. Once a message has been sent, `Send()` will block program execution until it receives a reply; this is known as *reply blocking*. The function takes five arguments:

**long id** This value uniquely identifies the receiver that the sender wants to send the message to. This is the value that is returned by a `Locate()` which is called prior to the `Send()`.

**void \*out** This is a pointer to memory containing the outgoing message intended for the receiver.

**int o** The size of the outgoing message in bytes.

**void \*in** This is a pointer to memory that will contain any incoming message replied by the receiver.

**int i** The maximum size of any incoming replied message from a receiver in bytes.

`Send()` returns the size of any replied message<sup>2</sup> or -1 in the case of a failure. Failures can occur due to:

- Program was interrupted during `Send`.
- The receiver identifier is incorrect.
- Some sort of failure in kernel communications.
- Failure to adjust kernel memory message buffer.
- Unable to add sender to sender table.
- Size of message to be sent is larger than the receiver's message buffer.

---

<sup>2</sup>In the case that the replied message size is zero, a NULL message was replied.

<b>int Trigger(long id, int proxy)</b>
--

The Trigger() call sends a proxy to a receiver. Unlike Send() though, Trigger() does not block awaiting a reply. The function takes two arguments:

**long id** This value uniquely identifies the receiver that the sending trigger program wants to send the message to. This is the value that is returned by a name\_locate() which is called prior to the Trigger().

**int proxy** The last argument is a proxy. A proxy is an integer  $\geq 1$ .

Trigger() returns 0 for success and -1 for failure. Failures can occur due to:

- Proxy value must be  $\geq 1$ , rejected otherwise.
- Program was interrupted during Trigger().
- The receiver identifier is incorrect.
- Some sort of failure in kernel communications.

Trigger() is sometimes used as "kicker" for a receiver. A receiver gets the proxy and then performs some predefined task without having to reply.

<b>int maxRecvSize(long id)</b>
---------------------------------

This function reports the maximum message size in bytes that a local receiver can take. The function takes one argument:

**long id** This value uniquely identifies the receiver.

This call returns  $\geq 0$  on success or -1 in the case of a failure.

<b>int maxRplySize(long id)</b>
---------------------------------

This function reports the maximum message size in bytes that a local sender can take in reply. The function takes one argument:

**long id** This value uniquely identifies the sender.

This call returns  $\geq 0$  on success or -1 in the case of a failure.

**int chkReceiver(long id)**

This function checks that a local receiver is still SRY name attached. The function takes one argument:

**long id** This value uniquely identifies the receiver.

This call returns 0 on success or -1 in the case of a failure.

**int chkSender(long id)**

This function checks that a local sender is still running. The function takes one argument:

**long id** This value uniquely identifies the sender.

This call returns 0 on success or -1 in the case of a failure.

**int rfd(void)**

The rfd() call returns the file descriptor used in conjunction with communicating with the kernel via device node */dev/sry*. This is particularly needed when a Receive() call is preceded by a poll() or select() call. The file descriptor required by these two system calls is provided by rfd().

**char \*sryErf(void)**

The sryErf() call returns a string, if set, that hopefully provides a clue when errors arise using SRY. Unfortunately, this only applies to problems arising at the application/library (user space) level.

**void sryLog(const char \*format, ...)**

This function writes a message in a *printf* format to the SRY log file. The SRY log file is */var/tmp/sry.log* by default. This call returns nothing.

### Q.3.2 User Space SRY Library

In subsection Q.3.1 we discussed each of the functions that comprise the *User Space SRY Library* and their purpose within application programs. These functions are then the connection between application programs and the SRY library. In this section we are going to look at how the connectivity between the SRY library and the kernel module is accomplished.

The connection between the SRY libraries and the kernel is the *system call*. The relevant system calls issued by applications using SRY and the SRY user space library are as follows:

1. open
2. close
3. poll
4. select
5. ioctl

The *open* system call is simply an open on the device node */dev/sry*. That is, `fd = open("/dev/sry", O_NOACCESS)`<sup>3</sup> which opens */dev/sry* for the above tabled system calls, but not for reading and writing. This provides a kind of safeguard against a program reading or writing on this device node. It returns a file descriptor used in making further system calls on this device node, mainly `ioctl()`.

The *close* system call releases the file descriptor returned by a preceding *open* call such that the file descriptor no longer refers to the */dev/sry* device file. ie. `close(fd)`.

The *poll* and *select* system calls are used when an application is simultaneously looking at several file descriptors and needs to know when to service each one. These system calls are not made by the SRY library but by the applications themselves. Note that in general it is possible to disable support for poll, select, epoll, kqueue, and devpoll by setting the respective environment variable `EVENT_NOPOLL`, `EVENT_NOSELECT`, `EVENT_NOEPOLL`, `EVENT_NOKQUEUE` and `EVENT_NODEVPOLL`.

---

<sup>3</sup>A file can be opened with the `O_NOACCESS` flag (defined as 3). The calling program must have both read and write permissions on the file. The purpose is to stipulate that a file descriptor for ioctl use is required and that no reading or writing is going to occur.

Finally, we come to the *ioctl* system call. This call is used for just about everything to do with the SRY message passing and accordingly, there will be much more said about it in what follows.

Table Q.3 contains a list of the user space SRY library functions and the various system calls used by each. Note that not all of the library functions make system calls, but most do.

SRY Library Call	System Call
openSRY	open, close, ioctl
closeSRY	close, ioctl
Locate	open, ioctl
Receive	ioctl
Send	open, ioctl
Trigger	open, ioctl
Reply	ioctl
ReplyError	ioctl
chkReceiver	ioctl
maxRecvSize	ioctl
maxRplySize	ioctl

Table Q.3: Library Call - System Call Correspondence

In a SRY nutshell, the *open* system call opens a connection to the SRY kernel module via the device node */dev/sry*. The *open* returns a file descriptor which is used by all of the subsequent SRY system calls, mainly *ioctl*. By using this file descriptor which was opened on the SRY device node, the kernel ‘knows’ which module to access. Given the name of the system call issued by the application/SRY library, the kernel again ‘knows’ which of the *file operations methods* within the SRY module to invoke.

### Q.3.3 Kernel Space SRY Module

Each user space system call corresponds to a file operations method contained in the SRY module. Within the SRY module, there is a very important structure called the *File Operations Structure*. The file operations methods corresponding to the user space system calls are listed in that structure and that is how the connection is made by the kernel. It is by this structure that the kernel ‘knows’ which action to take pertaining to a particular system call. Table Q.4

shows the correspondence between the user space system calls and the related file operations methods within the SRY module.

System Call	File Operations Method
open	sry_open
close	sry_release
poll	sry_poll
select	sry_poll
ioctl	sry_ioctl

Table Q.4: System Call - File Operations Method Correspondence

We will discuss each of these table entries in turn.

### **sry\_open**

This is the method called by the kernel to handle an *open* system call made by the SRY library on the */dev/sry* device node. It really only exists in order to let the kernel associate the relevant SRY driver functionality to a file descriptor.

### **sry\_release**

This is the method called by the kernel to handle a *close* system call made by the SRY library. It only exists so that the file descriptor associated with a prior *open* system call can be released and no longer pertains to the SRY driver module.

### **sry\_poll**

This method is called by the kernel to deal with a *poll* or *select* system call. It works in conjunction with *Send* and *Trigger*. The poll and select are called from application programs, not by any function within the SRY user space library. The poll and select system calls are generally used by application receiver programs which are monitoring more than one file descriptor at a time. ie. The file descriptor associated with */dev/sry* and some other like *stdin*. Interestingly, though a small method by design, parts of it apply to either poll or select, but not to both.

**sry\_ioctl**

This is the file operations method that takes care of almost everything to do with SRY. Observe Table Q.3, the *ioctl* system call is issued by every function in the SRY library and corresponds directly to the *sry\_ioctl* file operations method.

Table Q.5 shows the relationships between the SRY user space ‘C’ library functions and the relevant kernel module methods which are always preceded by **sry\_**. The connection is the user space call *ioctl* which in turn calls the kernel space operations method *sry\_ioctl*.

Observe “openSRY” as an example:

$$\text{openSRY} \iff \text{ioctl} \iff \text{sry_ioctl} \iff \text{sry_name_attach}$$

SRY Function Call	ioctl Command	SRY Module Call
openSRY	NAME_ATTACH	sry_name_attach
closeSRY	NAME_DETACH	sry_name_detach
Locate	NAME_LOCATE	sry_name_locate
Receive	RECEIVE	sry_receive
Send	SEND	sry_send
Trigger	TRIGGER	sry_trigger
Reply	REPLY	sry_reply

Table Q.5: SRY Library/Module Calls via ioctl

An alert reader might ask, “If all of these SRY library calls use the *ioctl()* system call, then how does the *sry\_ioctl()* method know who or what called it, and hence what action to take?”.

The answer to this question lies in the second column of Table Q.5. The actual source code layout of the *sry\_ioctl()* method is as follows:

```
static long sry_ioctl(struct file *f, unsigned cmd, unsigned long arg)
{
long ret = -1;

switch (cmd)
{
case NAMEATTACH:
    ret = sry_name_attach(arg);
    break;
```

```
case NAMEDETACH:
    ret = sry_name_detach(arg);
    break;

case NAMELOCATE:
    ret = sry_name_locate(arg);
    break;

case RECEIVE:
    ret = sry_receive(arg);
    break;

case SEND:
    ret = sry_send(arg);
    break;

case TRIGGER:
    ret = sry_trigger(arg);
    break;

case REPLY:
    ret = sry_reply(arg);
    break;

case RESET:
    ret = sry_reset();
    break;

case DUMP:
    ret = sry_dump();
    break;

case TASK:
    ret = sry_task(arg);
    break;

default:
    printk(KERN_NOTICE "Unknown ioctl cmd.\n");
}

return ret;
}
```

Essentially, `sry_ioctl()` is simply a switch statement. The second argument (*cmd*) in the function call dictates from which library function the `ioctl()` system

call was made and hence, what action to take. The third argument called *arg* is the connection for transferring data to and from kernel space and user space, ie. the SRY module and SRY ‘C’ or C++ libraries.

That leaves us with the third column of Table Q.5. These are the actual functions that perform all of the work within the SRY kernel module.

## SRY Kernel Module Notes

Rather than include the ‘C’ code that makes up the SRY kernel module and explain it ad nauseum, let’s describe the design with a more general point of view in mind. SRY is at its essence a system for **synchronous message passing**. In the next few paragraphs we will discuss what we mean by *synchronous* and *message passing*.

Let’s begin by looking at a typical SRY scenario and demonstrate what *synchronous* means. For simplicity suppose we have a process that is a receiver only by nature. This typically means that once the program starts it finds its way into an infinite loop with a `Receive()` call at the top of the loop. It will sit at the `Receive()` call and wait for a message to arrive. When a message arrives, the message content will be examined and the appropriate action taken, including a call to `Reply()`. While the program sits in `Receive()` and waits for a message to arrive, we say that the program is *receive blocked*. At the module level, `sry_receive()` puts the process in a *wait state*. When a message does finally arrive, the receiver is woken up, collects its message and other information and then returns. This is the point at which the user space `Receive()` returns and the message can be dealt with.

Let’s look at it from the point of view of the program that is sending the message. Again, for simplicity we will take the case of a pure sender; ie. a program that only sends messages. Initially, this program starts up and among other thing, name locates the receiver program it is interested in sending a message to. The name locating returns a unique SRY identifier to the receiver program. Using this in its `Send()`, the sender then sends the message to the receiver program. Within the SRY module, `sry_send()` tries to obtain the semaphore for the receiver. It will wait until the semaphore is available. The receiver semaphore might be available and it might not if the receiver is busy with another sender. This is the mechanism that provides the first instance of synchronous behaviour. Once the semaphore has been obtained, the sender will copy in its message and then awaken the receiver. At this point the sender will then enter a *wait state* until it is awoken. We refer to the sender as *reply blocked*.

Now, back to our receiver. It has received the message from the sender,

decided what to do with it and now replies back to the sender. It calls Reply() and transfers the reply message to the sender. At the sry\_reply() level, it awakens the sender and returns. At this point the Reply() then returns and we go around the receiver's infinite loop to return back Receive() where we await the next message.

The Send() in the sender program now returns and processes its reply message. The sequence of *wait-waken-process-wait* forms the core of the SRY synchronicity. Which thread is allowed to wake up another thread is controlled by contention over a semaphore on a first come first served basis. In this way, senders will be queued as they contend for a particular receiver's semaphore.

An explanation of the *message passing* aspect of SRY is our last topic for this section. Let's look at the simplest case of a sender passing a message to a receiver only. A sender application will compose a message in a way that is understood and agreed upon by both the sender and receiver; otherwise the receiver will not know how to interpret the message. The sender's outgoing message is contained in a memory buffer or a variable of some sort within the sender application. Accordingly, there exists a pointer to this memory location. The pointer is passed through the Send() library function to an ioctl() system call. From there, it is passed on to sry\_ioctl within or as the third argument.

Now the pointer (memory address) of the user space message is within the kernel space. From here the message contents are copied into a general memory buffer within the SRY module. At the proper time, the message contents are copied from this kernel space memory buffer into the receiver's user space message buffer. So, the message passing goes something like this:

- I A message is constructed and contained in the user space sender.
- II Send() is called with the address of the message.
- III The address is passed to ioctl() within Send().
- IV ioctl() passes the address to sry\_ioctl().
- V sry\_ioctl() calls sry\_send() which copies the message into a temporary kernel buffer.
- VI sry\_receive() copies the message from the temporary buffer into a buffer provided by Receive() via ioctl() and sry\_ioctl().
- VII The receiver now has the message within its user space memory buffer.

The copy from user space to kernel space and back to user space has been a source of some discomfort. Why not copy the data from the sender's user space directly to the receiver's user space? Clearly, this would be faster and there would be no need to look after a message buffer in kernel space. All true. However, if you have the opportunity to look through the routines that provide the copying service, you will see that there is very much to it. There are many checks and balances that depend on architecture, O/S version and so on. To maintain a separate set of functionality would be very unwise given the proliferation of Linux.

As well, the amount of memory available for message passing through the kernel is controlled ... there is a limit which at the time of this writing is five pages of memory. It pays to be careful of allowing this limit to be set too high for security reasons. The default memory size is one KB (1024 bytes). Messages larger than one KB and less than or equal to five pages of memory cause the SRY module to adjust its internal message buffer to be able to contain them.

## Q.4 Starting SRY

In order to enable SRY, the SRY module must inserted into the kernel. This is performed by a short script called (not surprisingly) *sry*. Note that running the SRY script to install the kernel module will not work if there is a SRY module currently being used. For example, say a receiver is running in the background, it must be killed before you can re-install the SRY module, otherwise the current SRY module will not be removed and then replaced.

At this time, the startup script for the SRY module is listed below.

```
1 #!/bin/sh
2
3 # definitions
4 MOD_DIR=/home/sry/src-mod
5 DEV_FILE_MODE=666
6
7 # remove current module (if present) quietly
8 rmmod sry 1> /dev/null 2>&1
9
10 # remove old device node (if present) quietly
11 rm -f /dev/sry 1> /dev/null 2>&1
12
13 # insert the module
14 insmod $MOD_DIR/sry.ko
```

```
15 sleep 1
16
17 # get the major number after the module insertion
18 major='cat /proc/devices | awk "\$2==\"sry\" {print \$1}"'
19 minor=0
20
21 # print the major and minor numbers to the screen
22 echo
23 echo "Major and Minor Numbers"
24 echo "-----"
25 echo "major number = $major"
26 echo "minor number = $minor"
27 echo
28
29 # check for kernel module
30 echo "lsmod check"
31 echo "-----"
32 echo "Module                      Size  Used by"
33 lsmod | grep sry
34 echo
35
36 # make the device node
37 mknod --mode=$DEV_FILE_MODE /dev/sry c $major $minor
38
39 # print the module information
40 echo "Module Information"
41 echo "-----"
42 modinfo $MOD_DIR/sry.ko
43 echo
```

lines 4-5 Set some useful variables.

line 8 Quietly remove the current SRY module, if present.

line 11 Quietly remove device node, if present.

line 14 Insert the SRY module into the kernel.

lines 18-19 Set the SRY device's major and minor numbers.

lines 22-34 Print to screen information regarding the SRY module/device.

line 37 Create the SRY device node or inode.

lines 40-43 Print to screen more information regarding the SRY module.

This script can be very convenient for development work on the SRY module software; it can also be called as part of a boot procedure.

## Q.5 SRY Development Directory Structure

Table Q.6 is a breakdown of the SRY development directory.

File/Directory	Purpose
Makefile	Upper level make
benchmarks	SIMPL, SRY-C and SRY-C++ benchmark programs
bin	executable programs
example	an SRY example
include	header files
lib	C and C++ static and dynamic libraries
pymodules	Python SRY modules
python	Python SRY development/tests
scripts	useful shell scripts
src-lib	C/C++ library development code
src-mod	'C' kernel module development code
surrogate	C++ TCP/IP surrogate development code
testing	C and C++ test code/programs
tools	test utility programs

Table Q.6: SRY Development Directory

We will discuss each of these table entries in turn. The order of the discussion will be governed by the order of the *general build* process.

### Q.5.1 Makefile

The top level make file simply calls all of the lower level make files within the development directory structure in the correct building order. Hence, it is responsible for the *general build* process. It can take one of three possible arguments:

1. clean
2. all

### 3. dynamic (default)

The *clean* argument simply tells all lower level make files to remove object, library and binary files, ie. start from ‘scratch’. The *all* argument tells all lower level make files to make everything in their particular area. The *dynamic* argument only applies to the library source code make file which will format the libraries for dynamic linking. It is the default, which is to say if ‘make’ is run without any command line arguments this will be the outcome.

## Q.5.2 Kernel Module Development Code

The directory **src-mod** contains the source code for the SRY kernel module. It is completely written in ‘C’. The make file can take either *clean* or *all* where *all* is the default target. There is only one source code file, namely *sry.c*.

## Q.5.3 C/C++ library Development Code

The directory **src-lib** contains the source code for the SRY user context ‘C’ and C++ libraries. There is only one source code file called *libsry.cpp*. In order to fool the ‘C’ compiler there is a symbolic link to the C++ source code file called *libsry.c*. This provides a ‘C’ only library and a C++ library from the same source code.

The make file can take either *clean*, *all/static* or *dynamic* where *dynamic* is the default target.

### Static vs. Dynamic

In referring to make files listed above, you may have noticed that some of the make file arguments define the targets *static* and *dynamic*. Having static and dynamic libraries available allows the programmer to decide which type of linking suits the situation best. While dynamically linked executables are somewhat smaller than those that are statically linked, the dynamic library must be present at the time of execution. If storage space is an issue, then only a numerical comparison of static vs. dynamic will decide which way is best. That is, the sum of the sizes of the statically linked executables compared to the sum of the sizes of the dynamically linked executables plus the size of the dynamic library itself.

### Q.5.4 Include

The **include** directory contains any needed header files. Note particularly the the *sry.h* header file which is required by any C/C++ applications that use SRY.

### Q.5.5 Libraries

The **lib** directory contains copies of the dynamic and static SRY libraries needed by any C/C++ application programs that use SRY.

### Q.5.6 Binaries

The **bin** directory contains any binary executables having to do with SRY. For example, the program that starts up the TCP/IP surrogates.

### Q.5.7 Shell Scripts

The **scripts** directory contains shell scripts which conveniently automate certain procedures. They are as follows:

- *sry*
- *start\_surrs*
- *stop\_surrs*

The *sry* script removes the SRY module from the kernel (if present) and then reloads the latest copy.

The *start\_surrs* script stops any currently running surrogates and then restarts them.

The *stop\_surrs* script stops any currently running surrogate programs.

The surrogate software enables one to seamlessly perform SRY on a network basis.

### Q.5.8 Tools

The **tools** directory contains programs which test and print out information regarding to SRY. They are written in C/C++. All of them run from the command line. They are as follows:

- *checkNode*

- dumpTables
- resetTables

The checkNode program simply tries to open a file descriptor to the /dev/sry inode. If the SRY module is present, it should work, otherwise it should fail.

The dumpTables program asks the SRY module to print out the receiver and sender table entries. This output can be seen by following the dumpTables command by the *dmesg* command.

The resetTables program asks the SRY module to clear its receiver and sender tables. This can be helpful for testing and developing. From a security standpoint though, it could be disastrous. Imagine that you have a number of programs sending and receiving from one and other only to have them cleared from the central control ... **hang and crash** describes the situation.

### Q.5.9 Testing

The **testing** directory contains programs which test out various aspects of SRY. These programs are broken into *local* and *network* tests and further into ‘C’ and C++ programs.

Another purpose of the various test programs is in providing simple examples of how the different members of the ‘C’ and C++ libraries are used and in varied circumstances.

Note that the *readme.C/CPPTests* file in this directory contains detailed descriptions of the various test programs and how they are used. Often, these programs are used together.

#### Local

There are two sets of programs with this subdirectory, one for ‘C’ and one with C++ in mind. Both sets are largely the same however, the API is somewhat different. The reason for this is to test the two available SRY libraries. The reason these programs are referred to as ‘local’ is because they run on the same host.

The following Table Q.7 provides a list of these local test programs.

Executable	Test
nameAttach	initialize a SRY receiver
nameLocate	locate a SRY receiver
receiver	receive a SRY message from sender or trigger
sender	send a SRY message to a receiver
senrec	receive SRY message from sender, send on to receiver
trigger	send a SRY proxy to a receiver
poller	poll on a SRY receiver
selector	select on a SRY receiver
syslog	test the SRY logging ability

Table Q.7: Local Test Programs

## Network

This subdirectory is laid out in the same fashion as the *local* directory mentioned above. There are fewer programs since it is unnecessary to test the same aspects a second time. These programs are referred to as ‘network’ because they are designed to send and receive their messages between different computer hosts via the surrogates.

**Note:** For testing purposes the same host computer may be used to run the receiving and sending programs by utilizing the loopback interface.

The following Table Q.8 provides a list of these network test programs.

Executable	Test
receiver	receive a SRY message from sender or trigger
sender	send a remote SRY message to a receiver
trigger	send a remote SRY proxy to a receiver

Table Q.8: Network Test Programs

### Q.5.10 TCP/IP Surrogate

The **surrogate** directory contains the source code for the TCP/IP surrogates. This surrogate software is responsible for allowing SRY between senders and

receivers that are not resident on the same host. They accomplish this via TCP/IP sockets. The reason for the name *surrogate* is both literal and practical. Suppose that a sender on computer host1 wants to send a message to a receiver on computer host2. When the sender performs its required Locate() it is given the SRY id of a replacement or surrogate receiver also on host1. It happily sends its message to the surrogate receiver and awaits the reply. The surrogate receiver then sends the message on via a TCP/IP socket connection to a surrogate sender residing on host2. This surrogate sender then SRY sends the message on to the original receiver and then awaits the reply. When the reply comes, the surrogate sender writes it back to the surrogate receiver on host1 which then SRY replies to the original sender.

The following Table Q.9 provides a list of parameters that may be changed through command line arguments when starting up the TCP/IP surrogates.

Parameter	Arg	Default
Checking Timeout	-c	60 seconds
Keep Alive Fail Limit	-f	3
Keep Alive Flag	-k	On
Memory Size Limit	-m	8192 bytes
Name Locate Timeout	-n	60 seconds
Communication Port	-p	50001

Table Q.9: TCP/IP Surrogate Command Line Arguments

The *Checking Timeout* refers to the amount of time in seconds between sending *keep alive* messages run by *surrogate\_r* on its *surrogate\_s* partner. If the *surrogate\_s* process does not reply to the *keep alive* message more times than the failure limit (see below), the *surrogate\_r* exits. At the same time, *surrogate\_r* also checks on the existence of the local sender that it is forwarding the messages for. If it is gone, again the *surrogate\_r* exits.

The *Keep Alive Fail Limit* is the number of times a *surrogate\_s* partner may fail to respond to *keep alive* messages.

The *Keep Alive Flag* will turn disable the *keep alive* checking. It is enabled by default.

The *Memory Size Limit* governs the size of the memory in bytes used to contain the messages sent, received, and replied.

The *Name Locate Timeout* is the amount of wait time in seconds allowed for a response to a name query. For example, suppose that a sender on host1 wishes

to send a message to a receiver on host2. The surrogates will negotiate the exchanges by first finding out whether firstly, host2 is reachable and secondly, if the receiver is SRY name attached on host2. Any appreciable delay in these interactions will cause the *Locate* call to fail.

The *Communication Port* is the port to which TCP sockets are attached in order to form SRY communications.

Suppose that the following was run from a terminal command line:

```
surrogate -c30 -f1 -m1024 -n30 -p50005 &
```

This starts the surrogates in the background, sets the keep alive checking to occur every 30 seconds, allows 1 keep alive failure, sets the message size upper limit to 1024 bytes, allows 30 seconds for a surrogate to be found on another host computer before reporting failure, and finally sets the port to 50005.

All of the TCP/IP based communications between surrogates is in a character format for all directives. The reason for this is to avoid the problems that might occur due to *structure padding* on binary structures. It's worth keeping in mind when designing your own messages to be sent over a network if they are based on 'C' structures. The unary operator *sizeof* may give you more than you bargained for.

Take for example the following code snippet:

```

1 typedef struct
2   {
3     short a;
4     int b;
5   } NEEE;
6
7 NEEE msgBuffer;
8 ...
9 Send( receiverId , &msgBuffer , sizeof(NEEE) , nullptr , 0 );
10 ...

```

Note the *Send* on line 9. The *sizeof(NEEE)* is given as 8 bytes on this computer. The data model being used on this computer would indicate that this structure should be 6 bytes, ie.  $2 + 4$ . The structure has been padded out to fit within memory boundaries. Eight bytes are going to be sent. How does the receiver interpret this message, especially if it is on a different platform and/or using a different data model? Care must be taken to plan your messaging across

platforms.

Another issue that may come up is network security. Utilities and programs such as *firewalls*, *selinux*, *kerberos* and so on may block messaging via SRY or SIMPL. You must be certain that sockets may be opened for reading and writing on the ports used by the surrogates.

The SRY surrogates have been extended to support IPv6 as well as IPv4 addressing. At the time of this writing SIMPL supports IPv4 only. Using the SRY *Locate* call with an IPv6 host name is now allowed. As a check, try using the loopback addresses *::1* and *127.0.0.1* as the host name in conjunction with the *sry/testing/network* programs to verify that your system supports IPv6 and/or IPv4 respectively.

### Q.5.11 Python SRY

The **python** directory contains the development software for a Python version of SRY. The Python SRY module that is developed here is only compatible with Python version 3 and not version 2. Note however, the SIMPL Python software is compatible with both versions 2 and 3.

Within this directory there is a Python script called **buildsrypython.py** which should be invoked using the Python 3.\* interpreter. This script interactively builds the SRY Python module called *sry.py* (not surprisingly) by searching the build host for a valid Python 3.\* source directory to build from and calling the top level Python make file. This script is also invoked by the top level SRY make file.

Now, the *sry.py* module is only partially written in Python. It is actually based on the SRY ‘C’ library through what are known in Python as ‘*C* extensions’. Aside from a speed advantage, it saves re-inventing that part of the wheel so to speak. These ‘C’ extensions are *hand wrapped* as opposed to being wrapped by a third party program such SWIG, Boost, etc. Several of these third party programs were tried with varying degrees of success. In the end it was decided to simply wrap the SRY ‘C’ functions manually.

The term *wrapping* indicates that the desired original SRY ‘C’ functions are included in a shared object ‘C’ library called *sryc.so* which is imported into *sry.py*. In turn, the Python scripts that are using SRY simply import *sry.py*. The development flow of the *sry.py* module is shown in Figure Q.2.

The *sry.py* module is very similar to the *csimpl.py* module in terms of functionality. However, the *csimpl.py* module allows programmers to exercise SIMPL functionality either from a procedural or an object oriented standpoint. The *sry.py* module takes an object oriented only point of view. Programs can ei-

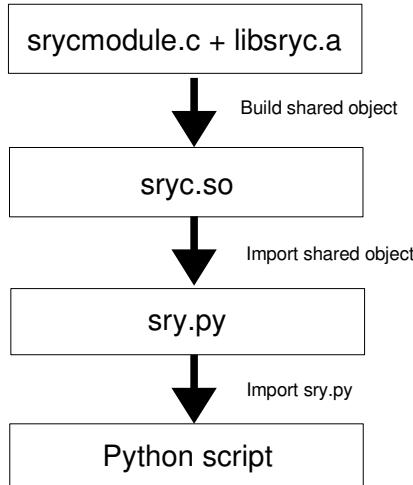


Figure Q.2: SRY Python

ther initiate a receiver (*Rec*) class and/or a sender (*Sen*) class object in order to utilize SRY.

Table Q.10 below contains a list of the SRY module *Sen* and *Rec* classes and methods.

Class	Method	Purpose
Rec	receive	Receive a message
Rec	reply	Reply to a message
Rec	returnProxy	Return a received proxy
Rec	rfd	Return the receive file descriptor
Sen	locate	Name locate a receiver
Sen	send	Send a message to a receiver
Sen	trigger	Send a proxy to a receiver
Sen	sfd	Return the send file descriptor
Rec/Sen	erf	Get the last SRY error message

Table Q.10: Python Sen and Rec Class Methods

Table Q.11 contains a list of the *Msg* class methods. This class provides the methods for composing, decomposing and generally manipulating the messages that are sent and received. *Msg* is contained within *Sen* and *Rec* as a base class.

Class	Method	Purpose
Msg	packMsg	Pack a message
Msg	packBool	Pack a C type boolean into a message
Msg	packChar	Pack a C type char into a message
Msg	packShort	Pack a C type short into a message
Msg	packInt	Pack a C type int into a message
Msg	packLong	Pack a C type long into a message
Msg	packFloat	Pack a C type float into a message
Msg	packDouble	Pack a C type double into a message
Msg	packString	Pack a C type string into a message
Msg	packBoolArray	Pack a C type boolean array into a message
Msg	packShortArray	Pack a C type short array into a message
Msg	packIntArray	Pack a C type int array into a message
Msg	packLongArray	Pack a C type long array into a message
Msg	packFloatArray	Pack a C type float array into a message
Msg	packDoubleArray	Pack a C type double array into a message
Msg	unpackMsg	Unpack a message
Msg	unpackBool	Unpack a C type boolean into a message
Msg	unpackChar	Unpack a C type char into a message
Msg	unpackShort	Unpack a C type short into a message
Msg	unpackInt	Unpack a C type int into a message
Msg	unpackLong	Unpack a C type long int into a message
Msg	unpackFloat	Unpack a C type float into a message
Msg	unpackDouble	Unpack a C type double into a message
Msg	unpackString	Unpack a C type string into a message
Msg	unpackBoolArray	Unpack a C type boolean array into a message
Msg	unpackShortArray	Unpack a C type short array into a message
Msg	unpackIntArray	Unpack a C type int array into a message
Msg	unpackLongArray	Unpack a C type long array into a message
Msg	unpackFloatArray	Unpack a C type float array into a message
Msg	unpackDoubleArray	Unpack a C type double array into a message

Table Q.11: Python Msg Class Methods

Note that all of the message packing/unpacking methods deal with ‘C’ data types with the char, short, int and long data types treated as signed.

**For concrete examples:** The top level Python make file also builds test

programs to exercise the *sry.py* module's capabilities. The Sen, Rec, and Msg class methods are all shown how to function in gory detail.

### Q.5.12 Python Modules

The **pymodules** directory contains the SRY Python modules that are imported into any Python scripts that are using SRY. This directory is usually exported to the environment variable called PYTHONPATH so that the Python SRY module called *sry.py* can be imported into any Python scripts. For example:

```
export PYTHONPATH=$PYTHONPATH:/home/sry/pymodules
```

Recall that Python scripts using *sry.py* may only be invoked by a Python 3 interpreter; version 2.7 is right out.

### Q.5.13 Benchmarking

The **benchmarks** directory contains three subdirectories: one for SIMPL , one for SRY ‘C’ and one for SRY C++. They each have an identical sender and an identical receiver with the exception that the SIMPL-based programs use SIMPL, whereas the SRY-based programs use SRY for their message passing. They even pass identical messages back and forth the same number of times. In terms of timing, SRY is not surprisingly faster by a factor of about three to ten times, depending on the architecture. One reason for this is the reduced context switching involved. This speed factor will of course vary depending on program loading and computer architecture.

It is also important to realize that these benchmarks are run on a single host. If they were run on different hosts it is thought that network slow downs will probably balance out the results more evenly.

To run the benchmarks, both SIMPL and SRY must be installed.

## Q.6 Example

In order to keep our example straightforward, there will be only one sender and one receiver. The sender will send a tokenized message to the receiver. For more on tokenized messages, see Chapter 5 on messaging.

**Receiver:** The SRY receiver is written in C/C++. Its function is to receive

a message and draw fractal images on a screen based on the message token and message contents. The receiver executes this task using Xwindows. The reason for this choice is speed. Within Xwindows, graphics calls are made via Xlib rather than Xt where possible, again for speed. It has been found that Xt runs just a little more slowly than direct calls to Xlib. Python was tried as a receiver/fractal generator because it supports complex numbers and is usually terrific for developing GUIs. By comparison with the C/C++/Xwindows receiver however, it was much much slower.

**Sender:** Python was chosen for the development of the SRY sender. This Python/Tk sender is a good choice for acquiring the information from the end user necessary to compose the message sent to the receiver.

To sum up, we have a Python/Tk SRY sender which provides a GUI and gathers the necessary message information. This message is sent to a C/C++/Xwindows SRY receiver. The receiver receives the message and following the information within, carries out its instructions.

For those readers interested in programming fractals, please see the various books on **Fractal Programming** by Roger T. Stevens. Many of the ideas and even some of the naming conventions used here originate from these books. The books are somewhat dated in that the code they use was written circa 1990 under DOS systems. That said, the mathematics has not changed nor has the intrinsic beauty of these fractals.

### Q.6.1 Fractals and the Example Programs

We would like to spend a few moments discussing fractals and why the example code that follows is written the way it is. Fractals are usually generated with the aid of *complex* numbers. As we will see a little further ahead, the equations used to generate each fractal in question are presented in terms of complex numbers. During development some of the fractals were generated using real numbers only as opposed to complex numbers. ie. treating the real and imaginary parts of the fractal generation separately. It was found that the real number splitting approach was faster than using the singular complex number approach. Hence, we worked strictly with a real number approach in our programming.

Seven types of fractals have been chosen for our example. These fractals are largely named in honour of the people who came up with them and/or worked on some aspect of them. They are as follows:

1. Mandelbrot
2. Julia
3. Cosine
4. Exponential
5. Legendre
6. Barnsley
7. Sierpinski

We will briefly discuss each one in turn. It's interesting to note that as a collection they act as a sort of secondary token in our SRY message. ie. How does the receiver program know which fractal to generate?

Each of the following are represented within the SRY receiver as eponymous methods. Each of the methods follow a similar format in that they all contain three nested loops. The first loop cycles through the rows of pixels of a graphical window. The second loop cycles through the columns of pixels of the graphical window. So, these two loops together cycle through a window one pixel at a time following a sort of Cartesian graphical display. Within these two loops we have the third loop which does the necessary algebra that determines the colour of that particular pixel.

Compare what is written below to the actual methods shown in the code for the SRY receiver, fractals.cpp, also printed below. The final equations given in each section are those that are used in our receiver program.

### Mandelbrot

$$z_{n+1} = z_n^2 + c \quad (\text{Q.1})$$

Equation Q.1 is used to generate the values needed to plot the Mandelbrot fractal image. It's intriguing that such a simple equation can be the source of such complexity, not to mention beauty. Given that  $z$  and  $c$  are complex numbers, we may express them in terms of real numbers as follows:

$$z_n = X(n) + Y(n)i \quad (\text{Q.2})$$

$$c = P + Qi \quad (\text{Q.3})$$

Substituting Q.2 and Q.3 into Q.1 we arrive at:

$$X_{n+1} = X_n^2 - Y_n^2 + P_n \quad (\text{Q.4})$$

$$Y_{n+1} = 2X_n Y_n + Q_n \quad (\text{Q.5})$$

In order to generate the Mandelbrot set we let  $z$  equal 0 and set the  $(x, y) = (\text{col}, \text{row})$  coordinates equal to the real and imaginary parts of  $c$  as they change over some selected range of values.

### Julia

$$z_{n+1} = z_n^2 + c \quad (\text{Q.6})$$

Equation Q.6 is used to generate the values required to plot the Julia fractal image. Note that it's the same equation as Q.1. The equations that follow are the same as in the Mandelbrot case except that in generating the Julia set we select a value of  $c$  and set the  $(x, y) = (\text{col}, \text{row})$  coordinates equal to the real and imaginary parts of  $z$  as it changes over some selected range of values.

### Cosine

$$z_{n+1} = \cos z_n + c \quad (\text{Q.7})$$

Note that:

$$\cos z_n = \cos(X_n + Y_n i) = \cos X_n \cosh Y_n - i \sin X_n \sinh Y_n \quad (\text{Q.8})$$

Substituting Q.2, Q.3, and Q.8 into Q.7 we arrive at:

$$X_{n+1} = P_n + \cos X_n \cosh Y_n \quad (\text{Q.9})$$

$$Y_{n+1} = Q_n - \sin X_n \sinh Y_n \quad (\text{Q.10})$$

### Exponential

$$z_{n+1} = c_n \exp(z_n) \quad (\text{Q.11})$$

Expanding:

$$\exp(z) = \exp(X + iY) = \cos Y(\sinh X + \cosh X) + i \sin Y(\sinh X + \cosh X) \quad (\text{Q.12})$$

Substituting Q.2, Q.3, and Q.12 into Q.11 we find:

$$X_{n+1} = (P_n \cos Y_n - Q_n \sin Y_n)(\sinh X_n + \cosh X_n) \quad (Q.13)$$

$$Y_{n+1} = (P_n \sin Y_n + Q_n \cos Y_n)(\sinh X_n + \cosh X_n) \quad (Q.14)$$

Equations Q.13 and Q.14 are iterated n+1 times until either the maximum number of iterations is achieved or certain dependent variables diverge. The colour of the pixel point in question depends on the last iteration of n. This is essentially the same approach used in all of the fractals we are using.

### Legendre

$$z_{n+1} = 0.125(35z_n^4 - 30z_n^2 + 3) + c \quad (Q.15)$$

Substituting Q.2 and Q.3 into Q.15 results in:

$$X_{n+1} = 0.125[35(X_n^2 - Y_n^2)(X_n^2 - Y_n^2) - 140X_n^2Y_n^2 - 30(X_n^2 - Y_n^2) + 3] + P_n \quad (Q.16)$$

$$Y_{n+1} = 2.5[7(X_n^2 - Y_n^2) - 3]X_nY_n + Q_n \quad (Q.17)$$

Equations Q.16 and Q.17 are used to determine loop limits and pixel colour.

### Barnsley

For the real part of z,  $x \geq 0$ :

$$z_{n+1} = z_n^2 - 1.0 \quad (Q.18)$$

For the real part of z,  $x < 0$ :

$$z_{n+1} = z_n^2 - 1.0 + PX_n \quad (Q.19)$$

Note that P and Q from Q.3 are such that P is a constant and Q = 0. Substituting Q.2 into Q.18 and Q.19:

For  $x \geq 0$ :

$$X_{n+1} = X_n^2 - Y_n^2 - 1.0 \quad (Q.20)$$

For  $x < 0$ :

$$X_{n+1} = X_n^2 - Y_n^2 - 1.0 + PX_n \quad (Q.21)$$

For all x,y:

$$Y_{n+1} = 2X_nY_n \quad (Q.22)$$

Which equation used for the next X is decided within the inmost loop that derives the pixel colour.

Sierpinski

For the real part of z,  $x < 0.5$ :

$$z_{n+1} = 2z_n - 1 \quad (\text{Q.23})$$

For the imaginary part of z,  $y < 0.5$ :

$$z_{n+1} = 2z_n - i \quad (\text{Q.24})$$

Otherwise:

$$z_{n+1} = 2z_n \quad (\text{Q.25})$$

Substituting Q.2 into Q.23, Q.24, and Q.25:

For the first case:

$$X_{n+1} = 2X_n - 1.0 \quad (\text{Q.26})$$

$$Y_{n+1} = 2Y_n \quad (\text{Q.27})$$

For the second case:

$$X_{n+1} = 2X_n \quad (\text{Q.28})$$

$$Y_{n+1} = 2Y_n - 1 \quad (\text{Q.29})$$

For the final case:

$$X_{n+1} = 2X_n \quad (\text{Q.30})$$

$$Y_{n+1} = 2Y_n \quad (\text{Q.31})$$

The algebra is very straightforward in this case. Each iteration simply doubles the previous value, checks the above inequalities, then possibly subtracts the number 1.

## Q.6.2 C/C++/Xwindows Receiver

### fractal.cpp

```

1 // system headers
2 #include <cstdlib>
3 #include <cmath>
4 #include <iostream>
5 #include <string>
6 #include <X11/Xlib.h>
7
8 // sry and application headers

```

```
9 #include <sry.h>
10 #include "fractals.h"
11 #include "fractalMsg.h"
12
13 using namespace std;
14
15 int main()
16 {
17 // display , window , gc , colourMap , winWid , winHgt are global
18
19 // X windows and local variables to main()
20 int screen;           // number of screen to place the window on
21 int depth;            // number of planes
22 unsigned maxDispWid; // width of the X display
23 unsigned maxDispHgt; // height of the X display
24 Window rwindow;       // root window
25 Visual *visual;       // colour info
26 XSetWindowAttributes attr; // for changing window attributes
27 long senderId;        // sender's sry id
28 int msgSize;           // size of received message
29 char msgBuffer[1024]; // message buffer
30 FRACTALMSG *msg;      // message template
31 const string me("FractalRenderer"); // sry receiver name
32 const string &mref = me; // sry receiver name reference
33
34 // instantiate a SRY object - name attach etc .
35 SRY comm(mref);
36
37 // open the connection to the display
38 display = XOpenDisplay(NULL);
39 if (display == NULL)
40 {
41     cerr << "Cannot connect to X server" << endl;
42     exit(EXIT_FAILURE);
43 }
44
45 // get the number of the default screen for our X server
46 screen = XDefaultScreen(display);
47
48 // get the plane depth
49 depth = XDefaultDepth(display , screen);
50
51 // get default colour map
52 colourMap = XDefaultColormap(display , screen);
53
54 // get the default graphics context
```

```
55 gc = XDefaultGC(display, screen);
56
57 // get maximal width of the default screen of X server in pixels
58 maxDispWid = XDisplayWidth(display, screen);
59
60 // get maximal height of the default screen of X server in pixels
61 maxDispHgt = XDisplayHeight(display, screen);
62
63 // get the default visual type
64 visual = XDefaultVisual(display, screen);
65
66 // set a black background for the window
67 attr.background_pixel = XBlackPixel(display, screen);
68
69 // find the ID of the root window of the screen, use as parent
70 rwindow = XRootWindow(display, screen);
71
72 // create a 'first' window as specified by prior calls visual etc.
73 window = XCreateWindow(
74     display, rwindow, 0, 0, 1, 1,
75     depth, InputOutput, visual, CWBackPixel, &attr);
76
77 while (true)
78 {
79     // get a message
80     msgSize = comm.Receive(&senderId, msgBuffer, sizeof msgBuffer);
81     if (msgSize == -1)
82     {
83         cerr << "Failed receive: " << strerror() << endl;
84         exit(EXIT_FAILURE);
85     }
86
87     // null reply - unblock Send()
88     msgSize = comm.Reply(senderId, nullptr, 0);
89     if (msgSize == -1)
90     {
91         cerr << "Failed reply: " << strerror() << endl;
92         exit(EXIT_FAILURE);
93     }
94
95     // line up on the message buffer
96     msg = (FRACTALMSG *)msgBuffer;
97
98     // what to do next?
99     if (msg->actionType == QUIT)
100    {
```

```

101    XDestroyWindow( display , window ); // close the window
102    XCloseDisplay( display ); // close connection to X server
103    exit( EXIT_SUCCESS ); // quit
104 }
105 else if ( msg->actionType == DRAW )
106 {
107     // close last window
108     XDestroyWindow( display , window );
109
110     // check and set window height and width
111     winWid = ( msg->maxCols < maxDispWid ) ? msg->maxCols : maxDispWid;
112     winHgt = ( msg->maxRows < maxDispHgt ) ? msg->maxRows : maxDispHgt;
113
114     // create the window as specified by prior calls visual etc.
115     window = XCreateWindow(
116         display , rwindow , 0 , 0 , winWid , winHgt , 1 ,
117         depth , InputOutput , visual , CWBackPixel , &attr );
118
119     // realize window and raise above other windows
120     XMapWindow( display , window );
121     XMapRaised( display , window );
122
123     Chaos *frac = new Chaos( msg ); // Chaos object draws fractal
124     frac->grid();
125     XFlush( display );
126     delete frac;
127 }
128 else
129     cerr << "unknown token type" << endl;
130 }
131
132 return 0;
133 }
134
135 void Chaos::grid( void )
136 {
137     // display , colourMap , colourName are global
138     XColor eColour; // exact colour: dummy var setting draw colour
139
140     // set the rgb's for the chosen colour scheme in colourName
141     for ( unsigned i = 0; i < m->maxColours; ++i )
142         // set the colours based on the names
143         XAllocNamedColor( display , colourMap , colourName[ m->colourIndex[ i ] ] ,
144             &eColour , &cColour[ i ] );
145
146     // deltas are the scaling of so many pixels per X and Y

```

```
147 deltaX = (m->XMax - m->XMin) / m->maxColumns;
148 deltaY = (m->YMax - m->YMin) / m->maxRows;
149
150 P = m->P;
151 Q = m->Q;
152
153 // which fractal to generate?
154 switch (m->fractalType)
155 {
156     case MANDELBROT:
157         mandelbrot ();
158         break;
159
160     case JULIA:
161         julia ();
162         break;
163
164     case COSINE:
165         cosine ();
166         break;
167
168     case EXPONENTIAL:
169         exponential ();
170         break;
171
172     case LEGENDRE:
173         legendre ();
174         break;
175
176     case BARNSLEY:
177         barnsley ();
178         break;
179
180     case SIERPINSKI:
181         sierpinski ();
182         break;
183
184     default:
185         cerr << "unknown fractal type" << endl;
186     }
187 }
188
189 void Chaos::mandelbrot (void)
190 {
191     // display , window , gc are global
192     double XY;
```

```

193
194 // loop through all columns
195 for (unsigned col = 0; col < m->maxColumns; ++col)
196 {
197 // loop through all rows per column
198 for (unsigned row = 0; row < m->maxRows; ++row)
199 {
200 // initialize starting values
201 X = 0.0;
202 Y = 0.0;
203 colour = 0;
204
205 // do the math for pixel point (col, row)
206 do
207 {
208 Xsq = X * X;
209 Ysq = Y * Y;
210 XY = X * Y;
211 Y = XY + XY + Q;
212 X = Xsq - Ysq + P;
213 colour++;
214 }
215 while ((colour < m->maxIters) && ((Xsq + Ysq) < m->divergence));
216
217 // set a drawing colour for this particular pixel
218 XSetForeground(display, gc, cColour[colour % m->maxColours].pixel);
219 // draw this pixel at (col, row)
220 XDrawPoint(display, window, gc, col, row);
221 // next Q for each row in the column
222 Q -= deltaY;
223 }
224
225 // next P per column
226 P += deltaX;
227 // first Q in the column
228 Q = m->YMax;
229 }
230 }
231
232 void Chaos::julia(void)
233 {
234 // display, window, gc are global
235 double XY;
236
237 // loop through all columns
238 for (unsigned col = 0; col < m->maxColumns; ++col)

```

```
239  {
240  // loop through all rows per column
241  for (unsigned row = 0; row < m->maxRows; ++row)
242  {
243  // initialize starting values
244  X = m->XMin + col * deltaX;
245  Y = m->YMax - row * deltaY;
246  colour = 0;
247
248  // do the math for pixel point (col, row)
249  do
250  {
251  Xsq = X * X;
252  Ysq = Y * Y;
253  XY = X * Y;
254  Y = XY + XY + Q;
255  X = Xsq - Ysq + P;
256  colour++;
257  }
258  while ((colour < m->maxIters) && ((Xsq + Ysq) < m->divergence));
259
260  if (colour >= m->maxIters)
261  colour = (unsigned)((Xsq - Ysq) * 6.0) % m->maxColours + 1;
262  else
263  colour = 0;
264
265  // set a drawing colour for this particular pixel
266  XSetForeground(display, gc, cColour[colour].pixel);
267  // draw this pixel at (col, row)
268  XDrawPoint(display, window, gc, col, row);
269  }
270 }
271 }
272
273 void Chaos::cosine(void)
274 {
275 // display, window, gc are global
276
277 // loop through all columns
278 for (unsigned col = 0; col < m->maxColumns; ++col)
279 {
280 // loop through all rows per column
281 for (unsigned row = 0; row < m->maxRows; ++row)
282 {
283 // initialize starting values
284 X = 0.0;
```

```

285     Y = 0.0;
286     colour = 0;
287     P = m->XMin + col * deltaX;
288     Q = m->YMax - row * deltaY;
289
290     // do the math for pixel point (col, row)
291     do
292     {
293         Xsq = X * X;
294         Ysq = Y * Y;
295         X = P + cos(X) * cosh(Y);
296         Y = Q - sin(X) * sinh(Y);
297         colour++;
298     }
299     while ((colour < m->maxIters) && ((Xsq + Ysq) < m->divergence));
300
301     // reset the colour?
302     colour=(colour>=m->maxIters) ? 0 : colour % (m->maxColours-1)+1;
303     // set a drawing colour for this particular pixel
304     XSetForeground(display, gc, cColour[colour].pixel);
305     // draw this pixel at (col, row)
306     XDrawPoint(display, window, gc, col, row);
307     }
308 }
309 }
310
311 void Chaos::exponential(void)
312 {
313     // display, window, gc are global
314
315     // loop through all columns
316     for (unsigned col = 0; col < m->maxColumns; col++)
317     {
318         // loop through all rows per column
319         for (unsigned row = 0; row < m->maxRows; row++)
320         {
321             // initialize starting values
322             X = 0.0;
323             Y = 0.0;
324             colour = 0;
325             P = m->XMin + col * deltaX;
326             Q = m->YMax - row * deltaY;
327
328             // do the math for pixel point (col, row)
329             do
330             {

```

```
331     Xsq = X * X;
332     Ysq = Y * Y;
333     X = (P * cos(Y) - Q * sin(Y)) * (sinh(X) + cosh(X));
334     Y = (P * sin(Y) + Q * cos(Y)) * (sinh(X) + cosh(X));
335     colour++;
336 }
337 while ((colour < m->maxIters) && ((Xsq + Ysq) < m->divergence));
338
339 // reset the colour?
340 colour=(colour >= m->maxIters) ? 0 : colour%(m->maxColours-1)+1;
341 // set a drawing colour for this particular pixel
342 XSetForeground(display, gc, cColour[colour].pixel);
343 // draw this pixel at (col, row)
344 XDrawPoint(display, window, gc, col, row);
345 }
346 }
347 }
348
349 void Chaos::legendre(void)
350 {
351 // display, window, gc are global
352 double diff, XY;
353
354 // loop through all columns
355 for (unsigned col = 0; col < m->maxColumns; col++)
356 {
357 // loop through all rows per column
358 for (unsigned row = 0; row < m->maxRows; row++)
359 {
360 // initialize starting values
361 X = 0.0;
362 Y = 0.0;
363 colour = 0;
364 P = m->XMin + col * deltaX;
365 Q = m->YMax - row * deltaY;
366
367 // do the math for pixel point (col, row)
368 do
369 {
370     Xsq = X * X;
371     Ysq = Y * Y;
372     diff = Xsq - Ysq;
373     X = 0.125 * (35*diff*diff - 140*Xsq*Ysq - 30*diff + 3) + P;
374     XY = X * Y;
375     Y = 2.5 * (7 * diff - 3) * XY + Q;
376     colour++;
```

```

377     }
378     while (( colour < m->maxIters) && (( Xsq + Ysq) < m->divergence));
379
380     // reset the colour?
381     colour=(colour >= m->maxIters) ? 0 : colour%(m->maxColours-1)+1;
382     // set a drawing colour for this particular pixel
383     XSetForeground(display , gc , cColour [ colour ].pixel );
384     // draw this pixel at (col , row)
385     XDrawPoint(display , window , gc , col , row );
386     }
387   }
388 }
389
390 void Chaos::barnsley(void)
391 {
392 // display , window , gc are global
393 double XY;
394
395 // loop through all columns
396 for (unsigned col = 0; col < m->maxColumns; col++)
397 {
398     // loop through all rows per column
399     for (unsigned row = 0; row < m->maxRows; row++)
400     {
401         // initialize starting values
402         X = m->XMin + col * deltaX ;
403         Y = m->YMax - row * deltaY ;
404         Xsq = 0;
405         Ysq = 0;
406         colour = 0;
407
408         // do the math for pixel point (col , row)
409         do
410         {
411             Xsq = X * X;
412             Ysq = Y * Y;
413             XY = X * Y;
414             X = (X >= 0) ? Xsq - Ysq - 1.0 : Xsq - Ysq - 1.0 + P * X;
415             Y = XY + XY;
416             colour++;
417         }
418         while (( colour < m->maxIters) && (( Xsq + Ysq) < m->divergence));
419
420         // reset the colour?
421         colour=(colour >= m->maxIters) ? 0 : colour%(m->maxColours-1)+1;
422         // set a drawing colour for this particular pixel

```

```
423     XSetForeground(display, gc, cColour[colour].pixel);
424     // draw this pixel at (col, row)
425     XDrawPoint(display, window, gc, col, row);
426 }
427 }
428 }
429
430 void Chaos::sierpinski(void)
431 {
432 // display, window, gc are global
433
434 // loop through all columns
435 for (unsigned col = 0; col < m->maxColumns; col++)
436 {
437     P = m->XMin + col * deltaX;
438
439     // loop through all rows per column
440     for (unsigned row = 0; row < m->maxRows; row++)
441     {
442         Q = m->YMax - row * deltaY;
443         X = P;
444         Y = Q;
445         colour = 0;
446
447         // do the math for pixel point (col, row)
448         do
449         {
450             X *= 2;
451             Y *= 2;
452             if (Y >= 1.0)
453                 Y -= 1;
454             else if (X > 1.0)
455                 X -= 1;
456             colour++;
457         }
458         while ((colour < m->maxIters) && (norm(X, Y) < m->divergence));
459
460         // reset the colour?
461         colour=(colour >= m->maxIters) ? 0 : colour%(m->maxColours-1)+1;
462         // set a drawing colour for this particular pixel
463         XSetForeground(display, gc, cColour[colour].pixel);
464         // draw this pixel at (col, row)
465         XDrawPoint(display, window, gc, col, row);
466     }
467 }
468 }
```

```
469  
470 inline double norm(double x, double y)  
471 {  
472     return x * x + y * y;  
473 }
```

lines 2-6 Include needed system header files.

lines 9-11 Include the SRY and application header files.

line 13 Use the standard namespace directly (saves some typing).

lines 20-26 Declare variables local to *main()* which are mainly needed for graphics.

lines 27-32 Declare variables needed for SRY. Note particularly the message buffer which is allocated 1K of memory and the *FRACTAL\_MSG* pointer which will act as a template over this memory buffer so that the message contents may interpreted correctly.

line 35 Instantiate an object of the SRY class. The object's name is *comm*. Note that the argument passed to the appropriate class constructor is the SRY name of this receiver.

lines 38-75 X window preparation.

lines 77-130 Receive an SRY message from the sender, carry out instructions according to the message token, null reply to the sender.

lines 80-85 Use the *Receive()* via the SRY object *comm*.

lines 88-93 The sender is sent a null reply because nothing is expected by the sender by way of a return message.

line 96 Lay the fractal message template over the message buffer so that it can be interpreted correctly. For simplicity it is assumed that this will be the only sort of message to be received.

lines 99-129 The receiver takes action based on the so-called *actionType* which is used as a token in the message. The available actions are to **quit** the program or to **draw** a particular fractal.

lines 105-127 The incoming message directs the receiver program to draw a fractal. The last window is destroyed and a new window is created with the correct sizing. A Chaos class object is instantiated and passed the message contents. Following the information in the message, a fractal image is drawn and when finished the Chaos object is deleted.

lines 135-187 The *grid()* method for the Chaos class is defined. It is code that is common to all of the fractals generated here.

lines 141-151 Perform some initialization that is needed by and common to all of the possible fractal displays.

lines 154-186 Call the relevant fractal display method based on which fractal type has been defined in the received message.

lines 189-230 Define the *mandelbrot()* method that is used for generating the eponymous fractal.

lines 232-271 Define the *julia()* method.

lines 273-309 Define the *cosine()* method.

lines 311-347 Define the *exponential()* method.

lines 349-388 Define the *legendre()* method.

lines 390-428 Define the *barnsley()* method.

lines 430-468 Define the *sierpinsk()* method.

Note that all seven fractal types are derived by the eponymous methods, hence a certain repetition. What is noteworthy in each case is the three nested loops and the algebra within the innermost loop. This algebra reflects the original complex numbered equation associated with the particular fractal and translated to real numbers. The outcome of the arithmetic decides pixel colour and the inmost loop limits.

Compare the code algebra within the inmost of the nested loops with the fractal by fractal explanation given in subsection Q.6.1 prior to this one.

The code for *fractal.h* is as follows.

### *fractal.h*

```

1 #ifndef _FRACTALS_H
2 #define _FRACTALS_H
3
4 #include "fractalMsg.h"
5
6 // chaos class definition
7 class Chaos
8 {
9     private:
10    FRACTALMSG *m;
11    double X, Y, Xsq, Ysq, deltaX, deltaY, P, Q;
12    unsigned colour;
13    XColor cColour[MAX_COLOURS]; // closest colour
14
15    void mandelbrot(void);
16    void julia(void);
17    void cosine(void);
18    void exponential(void);
19    void legendre(void);
20    void barnsley(void);
21    void sierpinski(void);
22
23 public:
24    Chaos(FRACTALMSG *p) {m = p;}
25    void grid(void);
26 };
27
28 // function prototypes
29 inline double norm(double, double);
30
31 // global variables
32 unsigned winWid;           // width for the new window
33 unsigned winHgt;            // height for the new window
34 Display *display;          // pointer to X Display structure
35 Window window;             // newly created window
36 GC gc;                    // graphics context
37 Colormap colourMap;        // the default colour map
38 const char *colourName[MAX_COLOURS] =
39     {"white",
40      "violet",
41      "blue",
42      "green",
43      "brown",

```

```
44     "yellow",
45     "orange",
46     "navy",
47     "cyan",
48     "sky blue",
49     "maroon",
50     "sea green",
51     "grey",
52     "black",
53     "azure",
54     "dark orange"};      // palette of 16 chosen colours
55
56 #endif
```

lines 7-26 Define the *Chaos* class.

lines 32-54 Define global variables mainly concerned with graphics.

The code for *fractalMsg.h* follows.

### *fractalMsg.h*

```
1 #ifndef _FRACTALMSG_H
2 #define _FRACTALMSG_H
3
4 // global constants
5 const unsigned MAX_COLOURS = 16;
6
7 // SRY bits and pieces
8 typedef enum
9 {
10     QUIT = 0,
11     DRAW
12 } ACTION_TYPE;
13
14 typedef enum
15 {
16     MANDELBROT = 0,
17     JULIA,
18     COSINE,
19     EXPONENTIAL,
20     LEGENDRE,
21     BARNSLEY,
22     SIERPINSKI
```

```

23 } FRACTAL_TYPE;
24
25 typedef struct
26 {
27 ACTION_TYPE actionType;
28 FRACTAL_TYPE fractalType;
29 unsigned maxCols;
30 unsigned maxRows;
31 unsigned maxIters;
32 double divergence;
33 double XMax;
34 double YMax;
35 double XMin;
36 double YMin;
37 double P;
38 double Q;
39 unsigned maxColours;
40 unsigned colourIndex [MAX_COLOURS];
41 } FRACTAL_MSG;
42
43 #endif

```

lines 5-23 Define constants used within the receiver program. Note the enumeration called *ACTION\_TYPE*; it is used as a token governing the action taken by the receiver.

lines 25-41 Define a structure containing the information needed by the receiver to accomplish its task. The structure members are as follows:

1. *actionType* - primary message token for action to take; draw or quit.
2. *fractalType* - secondary message token for which of the possible fractal images to draw.
3. *maxCols* - the width of the graphical window.
4. *maxRows* - the height of the graphical window.
5. *maxIters* - loop iteration limit.
6. *divergence* - loop iteration limit.
7. *XMax* - for display limits.
8. *YMax* - for display limits.
9. *XMin* - for display limits.

10. YMin - for display limits.
11. P - real part of a complex constant.
12. Q - imaginary part of the same complex constant.
13. maxColours - number of available colours.
14. colourIndex - the value of each available colour.

There will be more said about the various elements of the message structure when we discuss the SRY sender program below. The sender provides a Python 3.\* tkinter GUI where many of the values within the message can be modified and their effects noted.

### fractal.py

```
1 """
2 fractals.py
3
4 This gui program provides an interface for a user to enter various numbers into
5 a message, then sends this message to a C/C++-based program called fractals
6 which then produces a fractal graphical display.
7
8 The message passing between this program and the fractal generating program is
9 based on the C extension library for sry, ie. sry.py
10 """
11
12 # import necessary modules
13 import sry
14 import sys
15 from array import array
16 from tkinter import *
17 from tkinter.font import Font
18
19 global choice
20 global msg
21
22 def drawList(lst , cmd):
23     # Listbox creator general to this program
24     l = Listbox(root, selectmode=SINGLE, font='Helvetica 15 bold')
25     for act in lst:
26         l.insert(END, act)
27     l.grid()
28     l.config(width=0, height=0)
29     l.bind("<<ListboxSelect>>", cmd)
30
31 def select1(event): # from fractalList
32     value = event.widget.get(event.widget.curselection()[0])
33     event.widget.forget()
34     event.widget.destroy()
35
36 global choice
37 choice = value
38 if value == 'Quit':
39     quit()
```

```

40     else:
41         drawList(actionList, select2)
42
43 def select2(event): # from actionList
44     value = event.widget.get(event.widget.curselection()[0])
45     event.widget.forget()
46     event.widget.destroy()
47
48     if value == "Default":
49         setDefaultMsg()
50         sendMpacking(event)
51         drawList(fractalList, select1)
52     elif value == "Modify":
53         setDefaultMsg()
54         modifyFractal()
55     elif value == 'Back':
56         drawList(fractalList, select1)
57     else:
58         quit()
59
60 def modifyFractal():
61     # put up the window for modifying the default fractal message values
62     Label(root, text=choice, font='Helvetica 18 bold italic').grid(row=0)
63
64 """
65 Spinbox widget parameters saved in tuples, used for changing
66 the message member values sent to the fractal drawing program
67 (label, row, default setting, increment, low, high, format)
68 """
69 a0 = ('Columns: integer 50 to 2000', 1, str(msg[2]), 10, 50, 2000, "")
70 a1 = ('Rows: integer 50 to 1500', 2, str(msg[3]), 10, 50, 1500, "")
71 a2 = ('Iterations: integer 10 to 1000', 3, str(msg[4]), 1, 10, 1000, "")
72 a3 = ('Divergence: float 1.00 to 3000.00', 4, str(msg[5]), 0.01, 1, 3000,
73     "%.2f")
74 a4 = ('Xmax: float 0.1 to 10.0', 5, str(msg[6]), 0.1, 0.1, 10, "%.1f")
75 a5 = ('Ymax: float 0.1 to 10.0', 6, str(msg[7]), 0.1, 0.1, 10, "%.1f")
76 a6 = ('Xmin: float -10.0 to -1.0', 7, str(msg[8]), 0.1, -10, -1, "%.1f")
77 a7 = ('Ymin: float -10.0 to -1.0', 8, str(msg[9]), 0.1, -10, -1, "%.1f")
78 if choice == 'Julia':
79     a8 = ('P: float -1.000000 to 1.000000', 9, str(msg[10]), 0.000001, -1,
80           1, "%.6f")
81     a9 = ('Q: float 0.000000 to 1.000000', 10, str(msg[11]), 0.000001, 0,
82           1, "%.6f")
83 else:
84     a8 = ('P: float -5.0 to 5.0', 9, str(msg[10]), 0.1, -5, 5, "%.1f")
85     a9 = ('Q: float 0.0 to 1.0', 10, str(msg[11]), 0.1, 0, 1, "%.1f")
86 a = (a0, a1, a2, a3, a4, a5, a6, a7, a8, a9)
87
88 for index in range(10):
89     if index == 9 and choice == 'Barnsley': # no Q for Barnsley
90         continue
91     spinners(index, a[index])
92
93 b1 = Button(root, justify=CENTER, text='Back')
94 b1.grid(row=11, column=0)
95 b1.bind('<Button-1>', back)
96
97 b2 = Button(root, justify=CENTER, text='Send')
98 b2.grid(row=11, column=1)

```

```
99     b2.bind('<Button-1>', sendMpacking)
100
101 def spinners(index, a):
102     # put up the spin boxes to adjust message parameters
103     index += 2
104     l = Label(root, text=a[0], font='Helvetica 15 bold')
105     l.grid(row=a[1], sticky=W)
106     v = StringVar(root)
107     v.set(a[2])
108     s = Spinbox(root, increment=a[3], from_=a[4], to=a[5], wrap=True,
109                 font='Helvetica 15 bold', textvariable=v, format=a[6],
110                 command=(lambda: msgElement(index, s.get())))
111     s.grid(row=a[1], column=1)
112
113 def msgElement(index, value):
114     # ensure that value is what it should be
115     if index <= 4 or index == 12:
116         value = int(value)
117     if index >= 5 and index <= 11:
118         value = float(value)
119     global msg
120     msg[index] = value
121
122 def back(event):
123     for x in root.grid_slaves():
124         x.grid_remove()
125     drawList(actionList, select2)
126
127 def setDefaultMsg():
128     # something to start with
129     global msg
130     for index in range(7):
131         if choice == fractalList[index]:
132             for m in range(14):
133                 msg[m] = msgTup[index][m]
134             break
135
136 def sendMpacking(event):
137     # pack the message
138     nee.packMsg(nee.BIN, "iiiiiddddddII", msg[0], msg[1], msg[2],
139                 msg[3], msg[4], msg[5], msg[6], msg[7], msg[8], msg[9], msg[10],
140                 msg[11], msg[12], msg[13])
141
142 """
143 # alternate message packing scheme - good for checking one thing at a time
144 nee.packInt(msg[0], nee.BIN)
145 nee.packInt(msg[1], nee.BIN)
146 nee.packInt(msg[2], nee.BIN)
147 nee.packInt(msg[3], nee.BIN)
148 nee.packInt(msg[4], nee.BIN)
149 nee.packDouble(msg[5], nee.BIN)
150 nee.packDouble(msg[6], nee.BIN)
151 nee.packDouble(msg[7], nee.BIN)
152 nee.packDouble(msg[8], nee.BIN)
153 nee.packDouble(msg[9], nee.BIN)
154 nee.packDouble(msg[10], nee.BIN)
155 nee.packDouble(msg[11], nee.BIN)
156 nee.packInt(msg[12], nee.BIN)
157 nee.packIntArray(msg[13], nee.BIN)
```

```

158 """
159
160 # send the message to the SRY receiver - fractals.pp
161 if nee.send(receiverId) == -1:
162     print("send error-", nee.erf())
163
164 def quit():
165     # load and send a 'time to quit' message to the receiver
166     nee.packMsg(nee.BIN, "i", actions['QUIT'])
167     nee.send(receiverId)
168     sys.exit()
169
170 # operational start of the program ****
171
172 # gui Listbox lists
173 fractalList = ('Mandelbrot', 'Julia', 'Cosine', 'Exponential', 'Legendre',
174     'Barnsley', 'Sierpinski', 'Quit')
175 actionList = ('Default', 'Modify', 'Back', 'Quit')
176
177 # message tokens for the receiver - what to do and with whom
178 actions = {'DRAW':1, 'QUIT':0}
179 fractals = {'MANDELBROT':0, 'JULIA':1, 'COSINE':2, 'EXPONENTIAL':3,
180     'LEGENDRE':4, 'BARNESLEY':5, 'SIERPINSKI':6}
181
182 # colour indices
183 cIndex1 = [13,1,2,3,4,5,6,7,8,9,10,11,12,0,14,15]
184 cIndex2 = [13,8,2,3,4,5,6]
185
186 # default fractal message values as tuples
187 m0 = (actions['DRAW'], fractals['MANDELBROT'], 1600, 1000, 512, 4.0, 1.2, 1.2,
188     -2.0, -1.2, 1.2, len(cIndex1), array("I", cIndex1))
189 m1 = (actions['DRAW'], fractals['JULIA'], 1600, 1000, 96, 4.0, 1.8, 1.2, -1.8,
190     -1.2, -0.743036, 0.113467, len(cIndex2), array("I", cIndex2))
191 m2 = (actions['DRAW'], fractals['COSINE'], 1900, 1000, 64, 1000.0, 5.0, 8.0,
192     -7.0, -8.0, 0.0, 0.0, len(cIndex1), array("I", cIndex1))
193 m3 = (actions['DRAW'], fractals['EXPONENTIAL'], 1900, 1000, 64, 1000.0, 2.0,
194     1.0, 0.5, -1.0, 0.0, len(cIndex1), array("I", cIndex1))
195 m4 = (actions['DRAW'], fractals['LEGENDRE'], 1900, 1000, 64, 1000.0, 0.8, 0.4,
196     -0.8, -0.4, 0.0, 0.0, len(cIndex1), array("I", cIndex1))
197 m5 = (actions['DRAW'], fractals['BARNESLEY'], 1900, 1000, 64, 4.0, 1.7, 1.03,
198     -2.1, -1.03, 1.0, 0.0, len(cIndex1), array("I", cIndex1))
199 m6 = (actions['DRAW'], fractals['SIERPINSKI'], 1300, 1000, 64, 64.0, 1.3,
200     1.22, -0.5, -0.22, 0.0, 0.0, len(cIndex1), array("I", cIndex1))
201
202 # tuple of default message tuples - useful in loops
203 msgTup = (m0, m1, m2, m3, m4, m5, m6)
204
205 # message to be sent - a list so it can be modified
206 global msg
207 msg = [None] * 14
208
209 # make a sender instance of sry
210 nee = sry.Sen(1024)
211
212 # is there a host name?
213 if len(sys.argv) == 1:
214     host = ""
215 else:
216     host = sys.argv[1]

```

```
217 receiver = "FractalRenderer"
218
219 # name locate the sry receiver program
220 receiverId = nee.nameLocate(host, receiver, 1024)
221 if receiverId == -1:
222     print("name locate error-", nee.erf())
223     sys.exit()
224
225 # initialize Tk for graphics
226 root = Tk()
227
228 # bring up opening widget
229 drawList(fractalList, select1)
230
231 # handle user input
232 root.mainloop()
```

lines 13-17 Import the required functionality from the various modules. Note the *sry* module which provides the SRY capability.

lines 22-29 Define a general function that draws a list box for the GUI.

lines 31-58 Define functions that react to a list box selection.

lines 60-99 Define a function that creates a window that parents spin boxes which allow changes to the values of the message elements.

lines 101-134 Various functions for attending to small and frequent tasks.

lines 136-162 Definition of a function called *sendMpacking*. This function collates the data, loads the various message data elements and sends the message to the receiver. There are basically two ways of putting a message together; the short and the long. They are both included here. The first version has the advantage that it is indeed shorter. It follows a sort of a *C printf()* format. The longer version is more space consuming but it is clearer in its step by step packaging approach to a message. In the case of a complex message structure, it might be preferable.

lines 173-207 Various mutable and immutable objects are created. For example, default message contents take the form of tuples. The reason they are tuples is because they need not be changed after their creation. These values however are brought into the modify window and present a starting point. The outgoing message data is contained in a list which is a mutable object.

line 210 An SRY sender object called *nee* is created for the purpose of communicating with the receiver.

lines 213-217 A host name and an SRY name for the receiver are determined.

lines 220-223 The program does an SRY name locate on the receiver with the SRY name *FractalRenderer* that lives on computer host *host*.

line 226-232 Graphics are instantiated and the first window is displayed.

There is a real advantage in having the *fractals.cpp* program running on one host computer and the *fractals.py* program running on another host. When you send a message, default or altered, you can see the fractal image being drawn. If you have made any changes to the default message data, you can readily see the effects.

If you are using two separate hosts, the receiver host should have the best graphical ability and the sender host can really be anything as long as a Python 3 interpreter is available.

For two hosts the procedure is as follows:

#### For host 1:

1. Start SRY by running the script *sry* located in the */.../sry/scripts* directory.
2. Start the TCP/IP surrogates by running the script *start\_surrs* located in the same directory.
3. Run the fractal rendering program *fractals* located in the */.../sry/example/bin* directory.

#### For host 2:

1. Start SRY by running the script *sry* located in the */.../sry/scripts* directory.
2. Start the TCP/IP surrogates by running the script *start\_surrs* located in the same directory.
3. Run the fractal GUI program *fractals.py* located in the */.../sry/example/bin* directory. ie. **python3 fractals.py** Note that the SRY fractal receiver program must be running so that this program can SRY *nameLocate* it.

In terms of the host providing the GUI, the menu list shown in Figure Q.3 will be the first thing to appear after startup.

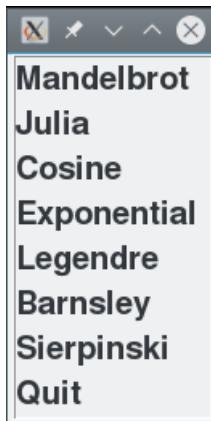


Figure Q.3: Menu One

The **Quit** at the bottom of the list causes the Python SRY sender to send a *quit* message to the SRY receiver. The receiver will exit, followed by the sender.

The first seven items on the list are our fractals. If one of these is chosen, then the menu list shown in Figure Q.4 will appear.



Figure Q.4: Menu Two

The **Quit** is the same as in the first menu. **Back** allows you to return to the first menu. **Default** sends a default or *canned* message to the receiver. This message concerns the fractal chosen in the first menu and contains general parameters. Upon receipt of this message, the receiver will draw the appropriate fractal within these general parameters. That brings us to the most interesting choice **Modify**. If this option is chosen, a frame with a series of list boxes as shown in Figure Q.5 will appear.

Near the top left of the window, the fractal of interest is listed; this was the fractal chosen in the first menu. On the bottom left, our old friend the **Back**

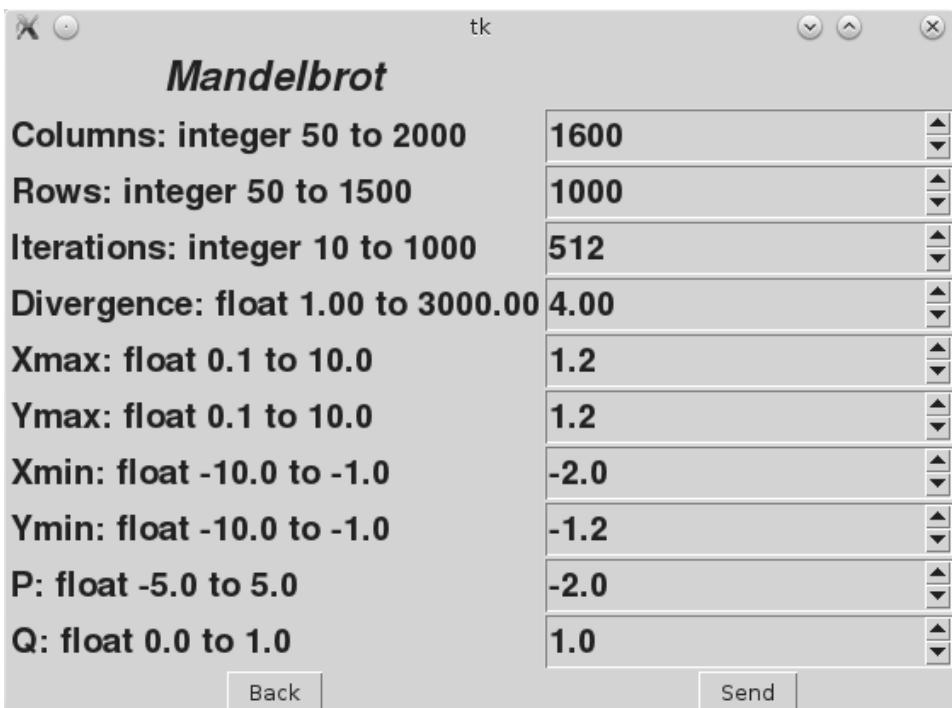


Figure Q.5: Menu Three

button is visible. On the bottom right, a button called **Send** is to be found. When you are finished with modifying the default message, select this button and the appropriate message will be sent to the SRY fractal receiver. The receiver will draw the fractal according to the changes that have been made. In this way, you can see the effects your changes make to the default fractal image.

Let's elaborate a little on the spin boxes. They start off by default with the original *canned* message values for each of the possible fractals. Their minimal and maximal values appear next to the quantity of interest along with the type of the variable.

1. Columns: integer 50 to 2000 - This is the width of the display window in pixels.
2. Rows: integer 50 to 1500 - This is the height of the display window in pixels.
3. Iterations: integer 10 to 1000 - This is a limit on how many times the inmost loop of the algorithm may run.

4. Divergence: float 1.00 to 3000.00 - This also is a limit on how long the program may remain within the inmost loop. The divergence represents a threshold that when crossed, indicates that the equations are blowing up towards infinity. This automatically signals that the pixel point in question will be black. Try adjusting this upward on some of the fractals and note that the images will become *less*.
5. Xmax: float 0.1 to 10.0 - A range variable. See the *fractal.cpp* code listing.
6. Ymax: float 0.1 to 10.0 - A range variable. See the *fractal.cpp* code listing.
7. Xmin: float -10.0 to -1.0 - A range variable. See the *fractal.cpp* code listing.
8. Ymin: float -10.0 to -1.0 - A range variable. See the *fractal.cpp* code listing.
9. P: float -5.0 to 5.0 - A constant (usually) of the generating equations. See the *fractal.cpp* code listing.
10. Q: float 0.0 to 1.0 - A constant (usually) of the generating equations. See the *fractal.cpp* code listing.

The following are the default fractal images.<sup>4</sup>

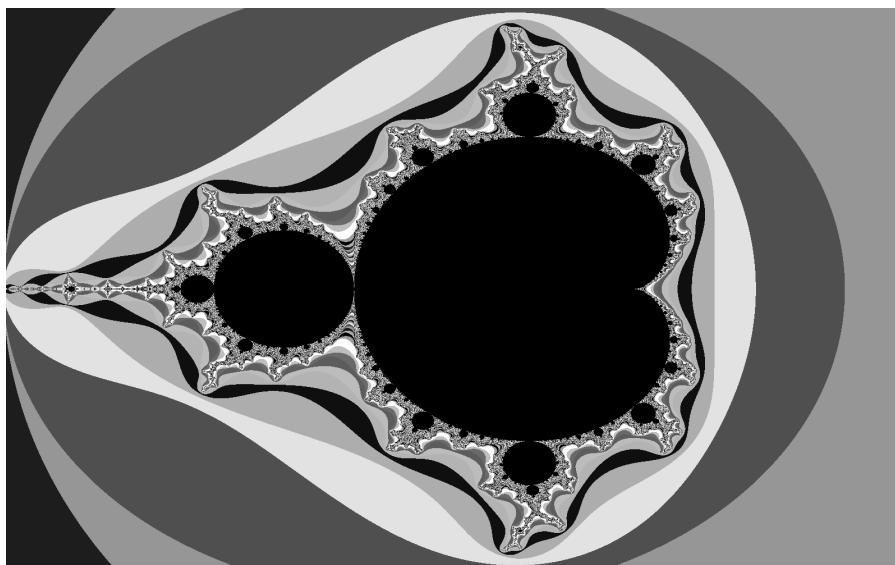


Figure Q.6: Mandelbrot Fractal

---

<sup>4</sup>Unfortunately we had to greyscale for the book; colour is much prettier!

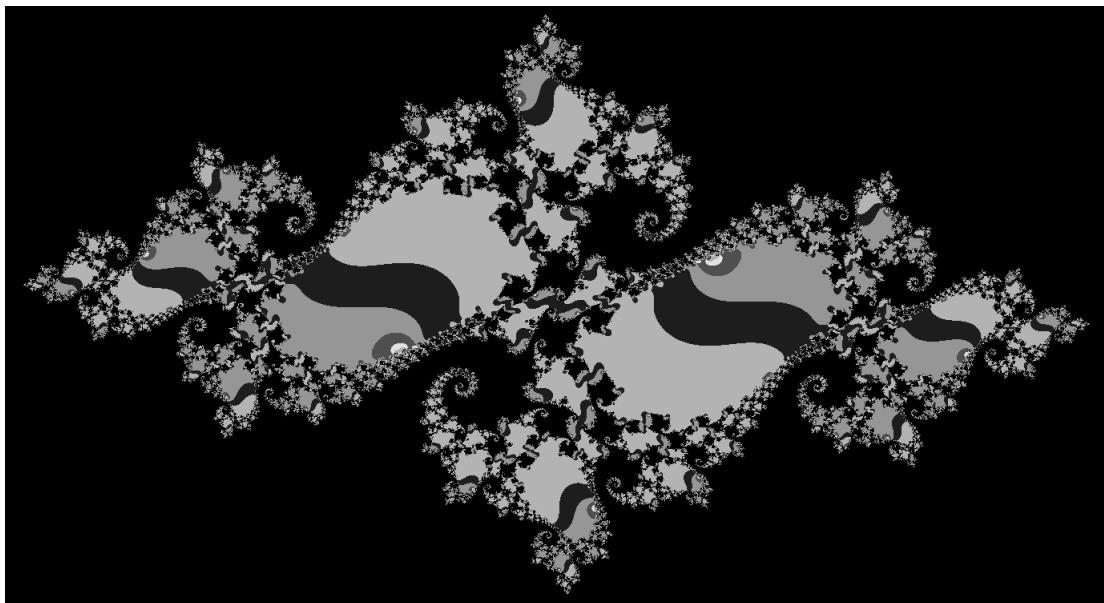


Figure Q.7: Julia Fractal

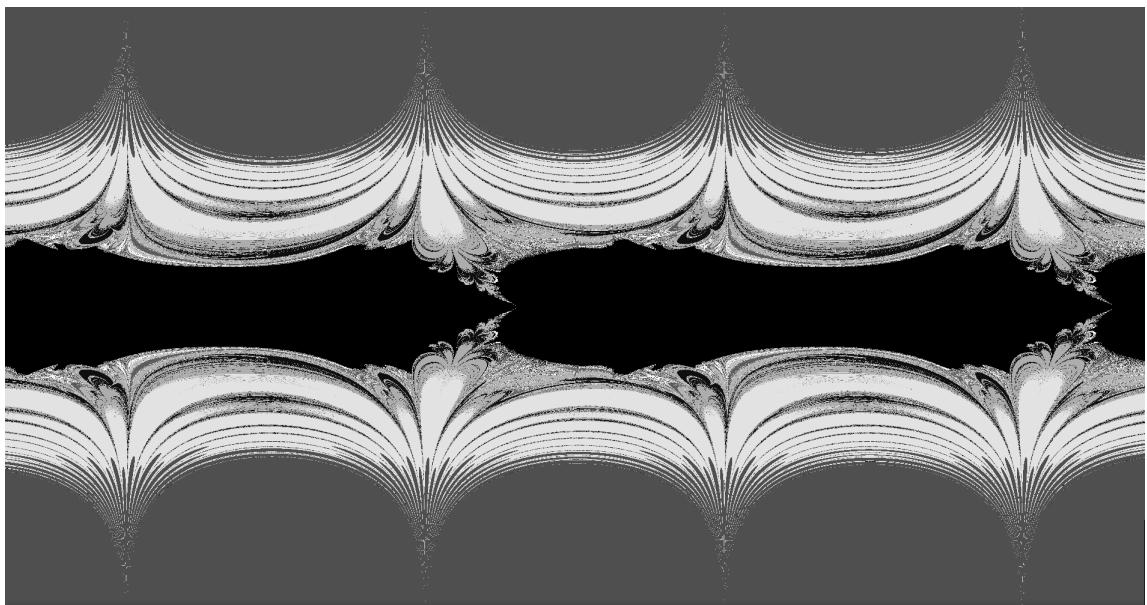


Figure Q.8: Cosine Fractal

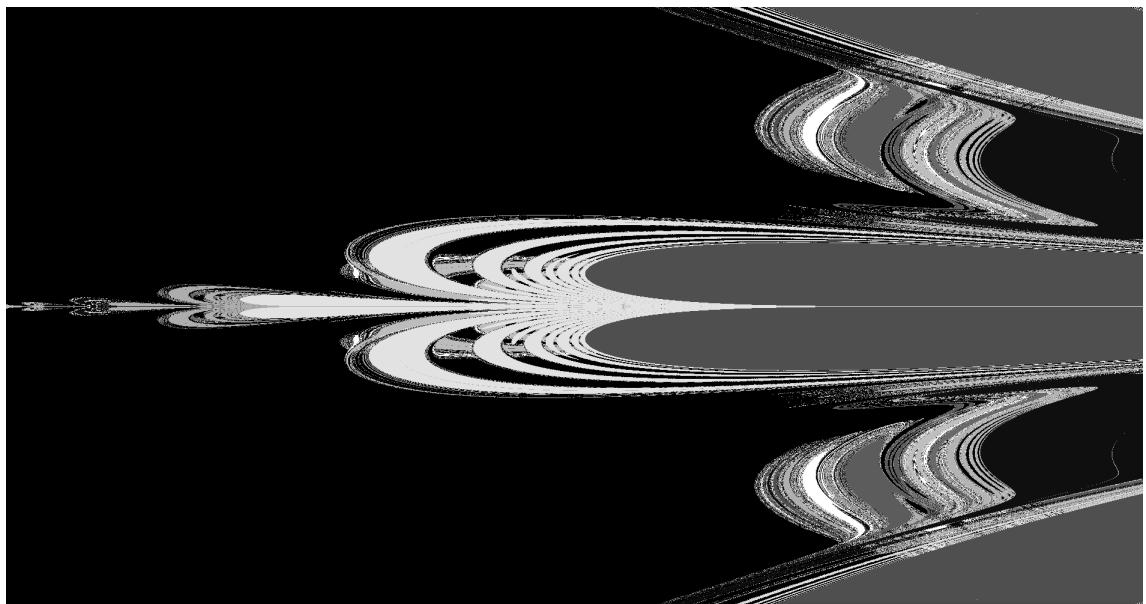


Figure Q.9: Exponential Fractal

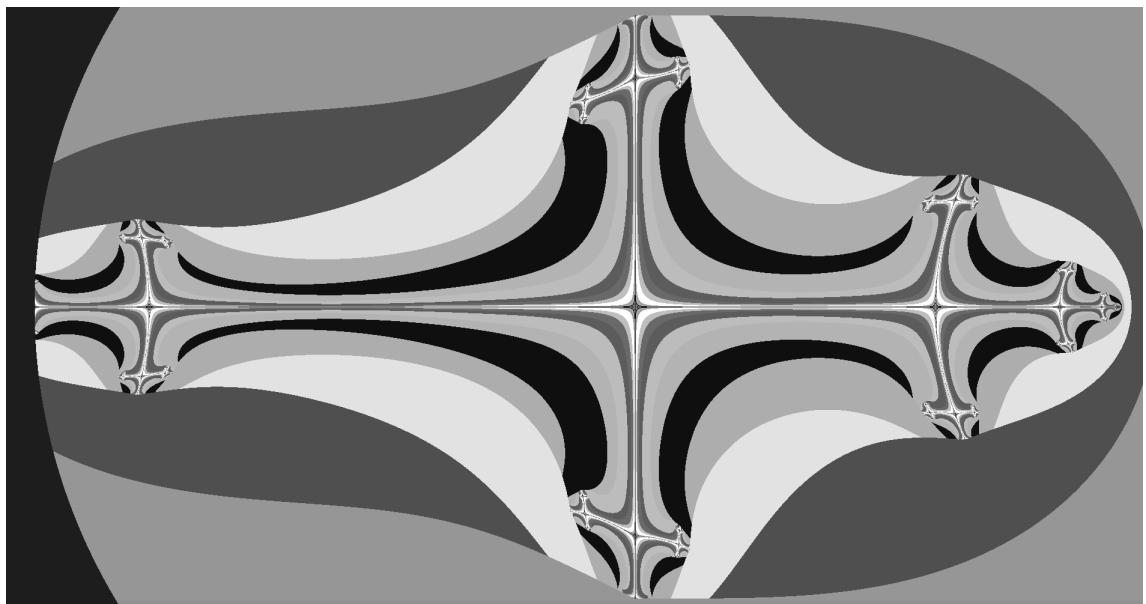


Figure Q.10: Barnsley Fractal

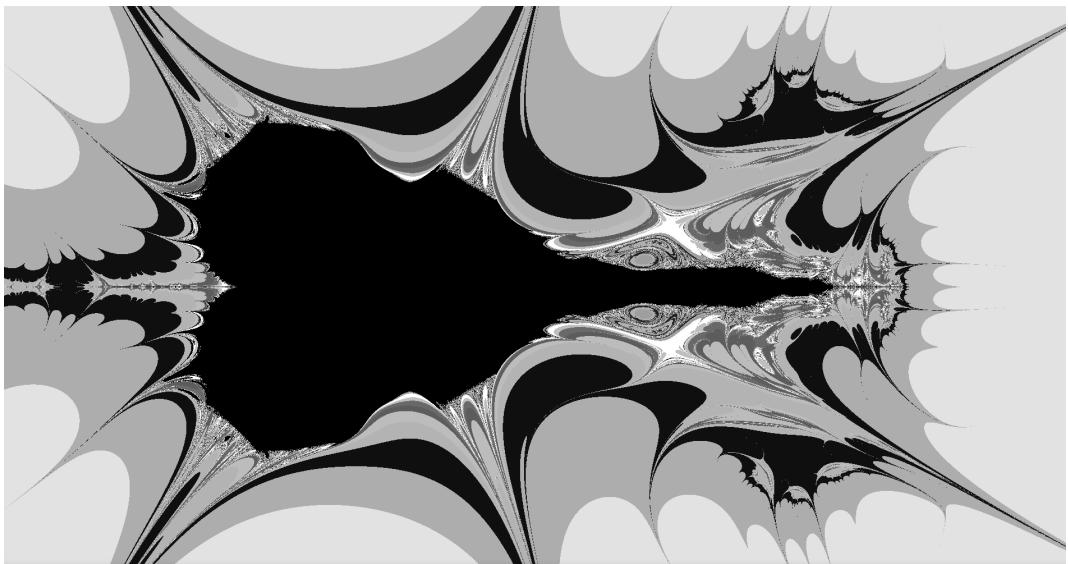


Figure Q.11: Legendre Fractal

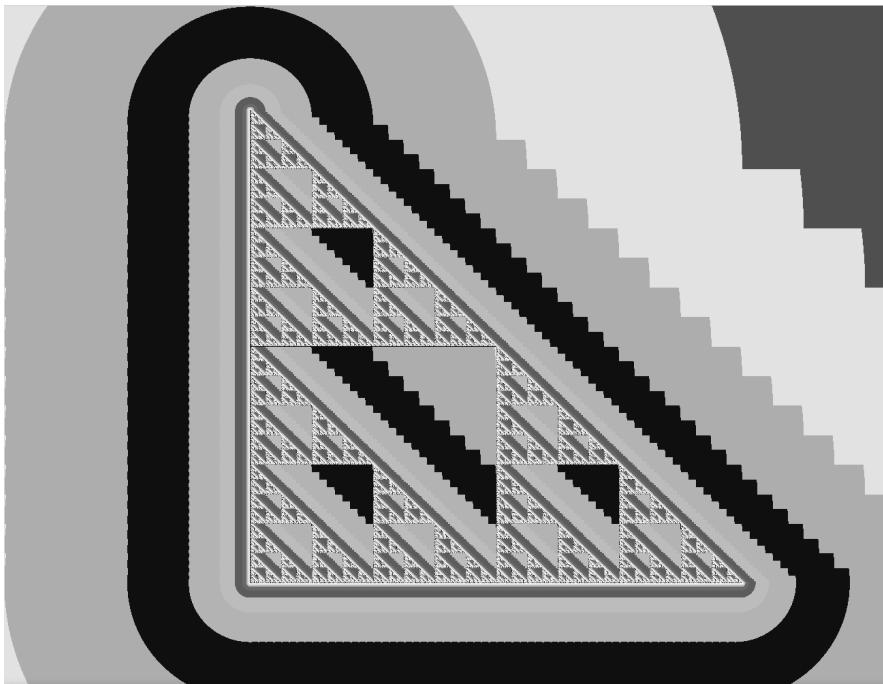


Figure Q.12: Sierpinski Fractal

## Q.7 Summary

In this appendix we covered a great deal of material. Some of the important points to keep in mind are:

- SRY differs from SIMPL in that local message passing is done with the aid of a kernel module. It is a mixture of kernel and user contexts whereas SIMPL is completely user context based. Given this, SRY is faster than SIMPL with respect to local messaging on single hosts.
- SRY can only run on later kernel versions of Linux because of its relying on a kernel module at its core. SIMPL was originally designed to run on Linux, MAC OS X, and most other UNIX-like operating systems. Hence, SIMPL can be more generally applied than SRY.
- With a few exceptions, the library API for function calls is the same between SIMPL and SRY. Note that SRY is often written in a C++ object format where name attaching is contained within the SRY class constructor.
- Currently, SRY application libraries only exist for C/C++ and Python version 3 whereas C/C++, Python versions 2 and 3, Tcl, and Java libraries currently exist for SIMPL.
- Surrogates are programs that occur in sender/receiver pairs and communicate with each other remotely. They live up their name by acting as replacements on different hosts for senders and receivers. As far as an application program using SRY is concerned, it is communicating with a local sender or receiver. Note that this is also the design employed by SIMPL surrogates.

## Q.8 Installing SRY

To install SRY, perform the following steps when you have obtained the SRY source code *sry.sh*.

1. Place *sry.sh* in /tmp.
2. Open a terminal in /tmp.
3. Run the following at the command line prompt: sh *sry.sh*

4. Answer any installation questions.

The file sry.sh contains the source code and an install script. Installing SRY in the /tmp means that you won't have to remove it if you don't want it. Essentially the installer aspect of sry.sh copies the SRY source into /tmp/sry. It then proceeds to build the software according to what is available on your host computer, installs the SRY module into the kernel, runs a benchmark test and then inquires if you wish to make a more permanent installation in some other part of your drive such as /home/sry for instance.

A description of the upper level make file can be found in section Q.5.1. Within the upper level make file is a call to a script that builds the Python 3 SRY modules. The name of the script is called *buildsrypython.py*. If you have a Python 3 development directory on your host computer, this script will try to find it and use it. If you have more than one, the script may find them and will prompt you to decide which of the available versions to use. If the script does not find any usable versions to use, it will simply print such and exit.

In the case of the Python SRY modules you may wish to add their home directory to the environment variable *PYTHONPATH*. For example, say the *sry* directory is in */home*. Adding the SRY Python modules to the PYTHONPATH environment variable would be something like:

```
export PYTHONPATH=$PYTHONPATH:/home/sry/pymodules
```

All of the building of the SRY bits and pieces is automated and should run if all of the C/C++ libraries and make facilities are present on the host computer.

# **Appendix R**

## **Glossary of Terms**

**ANSI** - American National Standards Institute.

**API** - Application Program Interface.

**CGI** - Common Gateway Interface.

**CPU** - Central Processing Unit.

**DHCP** - Dynamic Host Configuration Protocol.

**DSO** - Dynamic Shared Object.

**FIFO** - First In First Out; a named pipe.

**GNU** - GNU's Not Unix.

**GUI** - Graphical User Interface.

**HTML** - Hyper Text Markup Language.

**IC** - Integrated Circuit.

**ID** - IDentification.

**ID** - Integrated Development Environment.

**IØ** - InputOutput.

**IP** - Internet Protocol.

**IPC** - InterProcess Communication.

**JNI** - Java Native Interface.

**LGPL** - Lesser General Public License Agreement.

**OOP** - Object Oriented Programming.

**OS** - Operating System.

**PC** - Personal Computer.

**PCB** - Printed Circuit Board.

**PID** - Process IDentification.

**QA** - Quality Assurance.

**receiver** - A process which receives SIMPL messages.

**RTOS** - Real Time Operating System.

**SBC** - Single Board Computer.

**sender** - A process which sends SIMPL messages.

**SIMPL** - Synchronous Interprocess Message Project for Linux.

**SIMPL ID** - A unique integer representing a receiver which is returned by a *name\_locate* library function call and used by a sender to direct messages to said receiver.

**SIMPL System** - a collection of two or more processes that use SIMPL to communicate messages to each other.

**SIPC** - Synchronous InterProcess Communication.

**SoC** - System on a Chip.

**SRR** - Send/Receive/Reply.

**SRY** - Send/Receive/replY.

**STF** - SIMPL Testing Framework.

**surrogate** - A SIMPL process that uses a given protocol for communications between a sender and a receiver which reside on different host machines.

**TCP** - Transmission Control Protocol.

**TCP/IP** - Transmission Control Protocol/Internet Protocol.

**UI** - User Interface.

**URL** - Uniform Resource Locator.

**XML** - EXtensible Markup Language.

# Index

- \_ALLOC, 177
- ‘C’ Language Functions, 279
- Agency, 25, 89
- Apache, 251, 409
- applet, 88
- Arduino, 116, 320
- Arduino Functions, 303
- Broadcaster, 25, 91
- buildsimpl, 270
- C++, 20
- CGI, 251, 409
- cgi-bin, 409
- child\_detach() for ‘C’, 281, 301
- code reuse, 62
- Courier, 87
- crossCopy() for Python, 355
- csimpl, 319, 320
- datastore, 96
- dotest, 138, 140, 191
- dumpProtocolTable, 106
- endian issues, 111
- engine unit test (sb001), 202
- engine.c, 195
- engine.h, 194
- engineMsgs.h, 171
- engineProto.h, 195
- error messages, 421
- extendable, 62, 72, 169
- Extreme Programming, 135, 155
- fclogger, 412
- fcLogx(), 413
- FIFO\_PATH, 147, 269, 409
- forked\_engine.c, 220
- forking agent, 155
- forking engine, 219
- fractal.h, 474
- fractal.py, 477
- fractalMsg.h, 475
- fractals.cpp, 461
- getInBufferSize() for Java, 383
- getMsgSize(), 372
- getMsgSize() for Java, 372
- getOutBufferSize() for Java, 383
- getOutSizeAvailable() for Java, 383
- globalMask, 413
- gui.py, 206
- Hangman Game, 117
- host name, 105
- HTTP, 251
- httpd.conf, 409
- iCanProgram course, 11
- integration test, 159, 162, 215
- internal messages, 407
- interprocess communication (IPC), 5
- Java, 20
- Java Language Methods, 357

Java Native Interface, 357  
Java Packing Methods, 373  
Java Packing/Unpacking Methods, 372  
Java Unpacking Methods, 373  
Java/Swing GUI, 240  
Java/Swing GUI Sudoku startup script,  
    240  
JNI, 357  
Jimpl, 357, 358  
Jimpl() for Java, 358, 383  
  
keep alive, 396, 400  
KISS principle, 73  
  
Language Packages, 277  
LGPL, 3, 423  
loadOutBuffer() for Python, 355  
logit() for Tcl, 390  
  
make.out, 270  
  
name\_attach, 5, 29, 32, 33  
name\_attach for Tcl, 391  
name\_attach() for ‘C’, 279, 301  
name\_attach() for Tcl, 385  
name\_detach for Tcl, 391  
name\_detach() for ‘C’, 280, 301  
name\_detach() for Tcl, 386  
name\_locate, 5, 29, 32, 33  
name\_locate for Tcl, 391  
name\_locate(), 105  
name\_locate() for ‘C’, 283, 301  
name\_locate() for Tcl, 387  
nameAttach() for Arduino, 303, 317  
nameAttach() for Java, 360, 372  
nameAttach() for Python, 321, 322, 335  
nameDetach() for Arduino, 305, 317  
nameDetach() for Java, 361  
nameDetach() for Python, 322, 335  
nameDetach() for Java, 372  
  
nameLocate() for Arduino, 306, 317  
nameLocate() for Java, 363, 372  
nameLocate() for Python, 324, 335  
networked sudoku, 227  
NetX, 8  
  
Object Oriented Programming (OOP),  
    20  
  
packBool() for Python, 346  
packBoolArray() for Python, 346  
packByte() for Java, 375  
packByteArray() for Java, 375  
packChar() for Java, 375  
packChar() for Python, 346  
packCharArray() for Java, 375  
packDouble() for Java, 375  
packDouble() for Python, 346  
packDoubleArray() for Java, 375  
packDoubleArray() for Python, 346  
packFloat() for Java, 375  
packFloat() for Python, 346  
packFloatArray() for Java, 375  
packFloatArray() for Python, 346  
packInt() for Java, 375  
packInt() for Python, 346  
packIntArray() for Java, 375  
packIntArray() for Python, 346  
packLong() for Java, 375  
packLong() for Python, 346  
packLongArray() for Java, 375  
packLongArray() for Python, 346  
packMsg() for Python, 346  
packShort() for Java, 375  
packShort() for Python, 346  
packShortArray() for Java, 375  
packShortArray() for Python, 346  
packString() for Python, 346  
Polling Emitter, 97

port, 395  
posttest, 138, 145, 191  
pretest, 138, 191  
process name, 105  
protocol name, 105  
protocol router, 102, 106, 395, 399  
protocols, 107  
Proxy, 25, 82  
Public Domain License, 423  
Python, 10  
Python + engine integration test (sb004), 215  
Python GUI unit test (sb002), 211, 212  
Python Language Functionality, 319  
Python Packing Functions, 336  
Python Packing/Unpacking Functions, 335  
Python SIMPL, 274  
Python Unpacking Functions, 336  
Python/Tk, 205  
PYTHONPATH, 409  
  
QNX, 7, 22  
QNX surrogates, 111  
queue, 93, 98  
  
Raspberry Pi, 115  
Receive, 5, 51  
receive blocked, 32  
Receive for Tcl, 391  
Receive() for ‘C’, 286, 301  
Receive() for Arduino, 308, 317  
Receive() for Java, 366, 372  
receive() for Python, 325, 335  
Receive() for Tcl, 388  
receiver, 5, 25  
Relay, 25, 81  
Relay() for ‘C’, 290, 301  
remote name location, 102  
Reply, 5, 51  
reply blocked, 32, 88  
Reply for Tcl, 391  
Reply() for ‘C’, 289, 301  
Reply() for Arduino, 314, 317  
Reply() for Java, 370, 372  
reply() for Python, 330, 335  
Reply() for Tcl, 389  
returnInBuffer() for Python, 355  
returnOutBuffer() for Python, 355  
returnProxy() for ‘C’, 299, 301  
returnProxy() for Java, 371, 372  
returnProxy() for Python, 333, 335  
RS-232, 103  
RS-232 Surrogate, 397  
RS-232 surrogate, 108  
rs232\_rw, 397  
  
Scheduler, 93  
seed code, 165  
seetest, 139  
Send, 5, 29, 33  
Send for Tcl, 391  
Send() for ‘C’, 288, 301  
Send() for Arduino, 310, 317  
Send() for Java, 369, 372  
send() for Python, 331, 335  
Send() for Tcl, 388  
Send/Receive/Reply (SRR), 31, 34  
sender, 5, 25  
setDMs() for Python, 355  
setInByteOrder() for Python, 355  
setOutByteOrder() for Python, 355  
shared memory, 8, 30, 98, 101  
sim.c, 174  
sim.h, 173  
sim/stim unit test (sb000), 189, 191  
SIMPL, 3  
SIMPL benchmarks, 270

SIMPL sandbox, 147, 269  
SIMPL Testing Framework (STF), 6, 135, 137, 147, 159, 273  
Simpl() for Python, 335  
simpl2.so, 319  
simpl3.so, 319  
SIMPL\_HOME, 269  
SIMPLBOOK\_HOME, 165  
simlCheckProcess() for ‘C’, 299, 301  
simlRcopy() for ‘C’, 296, 301  
simlReplySize() for ‘C’, 297, 301  
simlScopy() for ‘C’, 296, 301  
simlSetSenderParms() for ‘C’, 298, 301  
simProto.h, 174  
simulator, 78, 172  
SIPC, 8, 425  
sockets, 395  
softwareIC, 4, 20, 169  
softwareICs repository, 272  
SRY, 425  
SRY Kernel Module, 425  
Ssimpl, 357, 358  
Ssimpl() for Java, 358, 383  
Stack, 85  
Statemachine, 95  
stim.c, 181  
stim.h, 180  
stimProto.h, 180  
stimulator, 79, 140, 141, 167, 172  
Sudoku, 159  
Sudoku engine, 193  
Sudoku via Web Browser, 252  
sudoku-cgi.py, 259  
sudoku.html, 253  
Sudoku.java, 241  
sudoku\_java, 240  
SUDOKU\_MSG, 171  
SUR\_ALIVE, 407  
SUR\_ALIVE\_REPLY, 407  
SUR\_CLOSE, 407  
sur\_detach() for ‘C’, 282, 301  
SUR\_DUMP\_TABLE, 407  
SUR\_ERROR, 407  
SUR\_NAME\_ATTACH, 407  
SUR\_NAME\_DETACH, 407  
SUR\_NAME\_LOCATE, 407  
SUR\_PROXY, 407  
SUR\_REPLY, 407  
SUR\_REQUEST\_PROTOCOL, 407  
SUR\_SEND, 407  
SUR\_SURROGATE\_READY, 407  
surDetach() for Java, 362, 372  
surDetach() for Python, 323, 335  
surrogate232, 397  
surrogate\_R, 393, 397  
surrogate\_r, 393, 397  
surrogate\_S, 395, 399  
surrogate\_s, 395, 399  
surrogates, 101, 107  
surrogateTCP, 393  
system picture, 154  
Tcl Language Procedures, 385  
Tcl SIMPL, 275  
Tcl/Tk GUI, 231  
Tcl/Tk GUI unit test (sb003), 232, 233  
tclsimpl, 320  
tclSurrogate, 9, 109, 320, 401  
tclSurrogate protocol, 401  
TCP/IP, 103  
TCP/IP Surrogate, 393  
TCP/IP surrogate, 108  
test plan, 159  
test.out, 203  
time out, 396, 400  
tokenized messages, 62  
top level Makefile, 166  
trace log, 143

trace logger, 411  
traceLogger for Tcl, 391  
Trigger for Tcl, 391  
Trigger() for ‘C’, 292, 301  
Trigger() for Java, 370, 372  
trigger() for Python, 332, 335  
Trigger() for Tcl, 389  
tunnel, 148  
typedef struct, 66  
  
unpackBool() for Python, 346  
unpackBoolArray() for Python, 346  
unpackByte() for Java, 375  
unpackByteArray() for Java, 375  
unpackChar() for Java, 375  
unpackChar() for Python, 346  
unpackCharArray() for Java, 375  
unpackDouble() for Java, 375  
unpackDouble() for Python, 346  
unpackDoubleArray() for Java, 375  
unpackDoubleArray() for Python, 346  
unpackFloat() for Java, 375  
unpackFloat() for Python, 346  
unpackFloatArray() for Java, 375  
unpackFloatArray() for Python, 346  
unpackInt() for Java, 375  
unpackInt() for Python, 346  
unpackIntArray() for Java, 375  
unpackIntArray() for Python, 346  
unpackLong() for Java, 375  
unpackLong() for Python, 346  
unpackLongArray() for Java, 375  
unpackLongArray() for Python, 346  
unpackMsg() for Python, 346  
unpackShort() for Java, 375  
unpackShort() for Python, 346  
unpackShortArray() for Java, 375  
unpackShortArray() for Python, 346  
unpackString() for Python, 346  
  
virtual machines, 148  
vwait, 238  
  
warning messages, 421  
web server, 251  
whatsMyError, 51  
whatsMyError(), 372  
whatsMyError() for ‘C’, 296, 301  
whatsMyError() for Arduino, 316, 317  
whatsMyError() for Java, 372  
whatsMyError() for Python, 333, 335  
whatsMyName() for ‘C’, 293, 301  
whatsMyReceiveFd() for Python, 333, 335  
whatsMyRecvfd() for ‘C’, 293, 301  
whatsMyRecvPtr() for ‘C’, 301  
whatsMyReplyfd() for ‘C’, 294, 301  
whatsMyReplyPtr() for ‘C’, 301  
whatsMyShmPtr() for ‘C’, 295  
whatsThisShmPtr() for ‘C’, 295  
Windows, 320





