

Contents

About the Author	xiii
Preface	xv
Acknowledgments	xix
1 Introduction	1
1.1 Requirements for Extreme Systems	2
1.1.1 Availability	2
1.1.2 Reliability	3
1.1.3 Scalability	4
1.1.4 Capacity	5
1.1.5 Productivity	5
1.2 Becoming Carrier Grade	7
1.3 Characteristics of Extreme Systems	9
1.3.1 Embedded	9
1.3.2 Reactive	9
1.3.3 Stateful	10
1.3.4 Real-Time	12
1.3.5 Distributed	12
1.4 Summary	13
2 Overview	15
2.1 Basic Terminology	15
2.2 Extreme System Reference Model	16
2.3 Programming Models	18
2.4 A Pattern Language for Carrier-Grade Software	19
2.5 Class Hierarchy	26
2.6 Summary	26
3 Object Orientation	27
3.1 Basic Principles	27
3.2 Choosing a Language	29

3.3	Other Considerations	30
3.4	Distractions and Myths	31
3.5	Summary	33
4	Using Objects Effectively	35
4.1	The Object Class	36
4.2	Basic Design Patterns	40
4.2.1	Singleton	40
4.2.2	Flyweight	41
4.2.3	Registry	41
4.2.4	Polymorphic Factory	42
4.2.5	Cached Result	46
4.3	Improving Availability	46
4.3.1	Object Pool	46
4.3.2	Object Nullification	51
4.3.3	Implementing Object Pools	51
4.4	Speeding Up Object Construction	57
4.4.1	Embedded Object	57
4.4.2	Object Template	60
4.4.3	Quasi-Singleton	61
4.4.4	Object Morphing	63
4.4.5	Implementing the Techniques for Pooled Objects	65
4.5	Summary	75
5	Scheduling Threads	77
5.1	Standard Scheduling Policies	77
5.2	Critical Regions	78
5.3	Problems with Standard Scheduling Policies	79
5.3.1	$p()$ and $v()$ Considered Harmful	79
5.3.2	Thread Starvation	80
5.3.3	Priority Inversion	80
5.4	Cooperative Scheduling	80
5.4.1	Implementing Thread Locking	81
5.4.2	I/O Threads and Invoker Threads	82
5.4.3	Run-to-Completion Timeout	83
5.5	Dealing with Blocking Operations	86
5.5.1	Supporting Blocking Operations by Locked Threads	87
5.5.2	Preventing Blocking Operations by Locked Threads	88
5.5.3	Prohibiting Synchronous Messaging	89
5.6	The Thread Class	91
5.7	Proportional Scheduling	95
5.7.1	Benefits of Proportional Scheduling	96
5.7.2	Implementing Proportional Scheduling	97
5.7.3	Approximating Proportional Scheduling	100
5.8	How Many Threads Do You Need?	101
5.9	Summary	103
6	Distributing Work	105
6.1	Reasons for Distribution	105
6.1.1	Increasing Capacity	105

CONTENTS

ix

6.1.2	Improving Survivability	106
6.1.3	Separating Programming Models	106
6.2	Complexities Caused by Distribution	107
6.2.1	Timeouts	107
6.2.2	Transient States	107
6.2.3	Increased Latency	108
6.2.4	Protocol Backward Compatibility	108
6.3	Techniques for Distribution	109
6.3.1	Heterogeneous Distribution	109
6.3.2	Homogeneous Distribution	113
6.3.3	Hierarchical Distribution	114
6.3.4	Symmetric Multi-Processing	115
6.3.5	Half-Object Plus Protocol	116
6.4	Summary	117
7	Protecting Against Software Faults	119
7.1	Defensive Coding	119
7.2	Protecting Against Trampers	120
7.2.1	Stack Overflow Protection	121
7.2.2	Low-Order Page Protection	122
7.2.3	Protecting Critical Data	123
7.2.4	User Spaces	123
7.2.5	Disadvantages of User Spaces	124
7.2.6	Write-Protected Memory	124
7.2.7	Protecting Objects	127
7.2.8	When Are User Spaces Appropriate?	128
7.3	Summary	130
8	Recovering from Software Faults	131
8.1	Safety Net	131
8.1.1	Catching Exceptions	132
8.1.2	Catching Signals	133
8.1.3	Fatal Errors	134
8.1.4	Recovering Resources	134
8.1.5	Implementing the <code>Thread</code> Class	136
8.2	Leaky Bucket Counter	155
8.3	Audit	157
8.3.1	Resource Pool Audit	158
8.3.2	Critical Regions	159
8.3.3	Queue Corruption	160
8.3.4	Traps in Callbacks	161
8.3.5	Distributed Resource Pools	163
8.4	Watchdog	163
8.5	Protecting Against Gobblers	164
8.6	Escalating Restarts	164
8.7	Initialization Framework	166
8.7.1	Registering Modules	166
8.7.2	Initializing a Node	167
8.7.3	Determining Module Initialization Order	168
8.7.4	Restarting a Node	168
8.7.5	Ensuring that Initialization Succeeds	169

8.7.6	Thread Observers	170
8.7.7	Binary Database	170
8.8	Summary	172
9	Messaging	175
9.1	Reliable Delivery	175
9.2	Message Attenuation	178
9.3	TLV Message	178
9.3.1	Parameter Typing	180
9.3.2	Parameter Fence	182
9.3.3	Parameter Template	182
9.3.4	Parameter Dictionary	187
9.4	Eliminating Copying	187
9.4.1	In-Place Encapsulation	187
9.4.2	Stack Short-Circuiting	188
9.4.3	Message Cascading	190
9.4.4	Message Relaying	191
9.4.5	Eliminating I/O Stages	193
9.5	Eliminating Messages	196
9.5.1	Prefer Push to Pull	196
9.5.2	No Empty Acks	197
9.5.3	Polygon Protocol	198
9.5.4	Callback	198
9.5.5	Shared Memory	200
9.6	Summary	201
10	Overload Controls	203
10.1	Finish What You Start	204
10.2	Discard New Work	206
10.3	Ignore Babbling Idiots	209
10.4	Throttle New Work	210
10.5	Summary	212
11	Failover	213
11.1	Redundancy Techniques	213
11.1.1	Load Sharing	214
11.1.2	Cold Standby	215
11.1.3	Warm Standby	216
11.1.4	Hot Standby	217
11.2	Checkpointing Techniques	217
11.2.1	Application Checkpointing	218
11.2.2	Object Checkpointing	218
11.2.3	Memory Checkpointing	219
11.2.4	Virtual Synchrony	219
11.3	Addressing Processors	221
11.4	Interacting with Escalating Restarts	222
11.5	Summary	223
12	Software Installation	225
12.1	Hitless Patching	226

CONTENTS

xi

12.1.1	Function Patching	226
12.1.2	Class Patching	227
12.1.3	Managing Patches	229
12.2	Hitless Upgrade	230
12.3	Rolling Upgrade	231
12.4	How Hitless Do You Need To Be?	232
12.5	Summary	233
13	System Operability	235
13.1	Administration	236
13.1.1	Configuration Parameters	236
13.1.2	Provisioning	237
13.2	Operations	238
13.2.1	Logs	238
13.2.2	Alarms	241
13.2.3	Operational Measurements	241
13.3	Maintenance	242
13.3.1	Handling Faults	243
13.3.2	High-Level Design	244
13.3.3	Commands	245
13.3.4	Hardware Requirements	247
13.3.5	Detecting Outages	247
13.4	Commercially Available Hardware and Middleware	249
13.5	Summary	249
14	Software Optionality	253
14.1	Conditional Compilation	254
14.2	Software Targeting	255
14.3	Run-Time Flags	256
14.4	Summary	256
15	Debugging in the Field	257
15.1	Software Logs	258
15.1.1	Software Error Log	259
15.1.2	Software Warning Log	261
15.1.3	Object Dump	262
15.2	Flight Recorder	262
15.3	Trace Tools	263
15.3.1	Function Tracer	263
15.3.2	Message Tracer	267
15.3.3	Tracepoint Debugger	268
15.4	Summary	268
16	Managing Capacity	271
16.1	Set the Capacity Benchmark	271
16.2	Measuring and Predicting Capacity	272
16.2.1	Transaction Profiler	272
16.2.2	Thread Profiler	276
16.2.3	Function Profiler	276
16.3	Summary	277

17 Staging Carrier-Grade Software	279
17.1 Selecting an Infrastructure	279
17.1.1 General Guidelines	280
17.1.2 Compiler	280
17.1.3 Operating System	281
17.1.4 Debugging	281
17.2 Adding Carrier-Grade Techniques	282
17.2.1 Techniques for Release 1.0	282
17.2.2 Techniques for Release 2.0 and Beyond	285
18 Assessing Carrier-Grade Software	287
18.1 Language	288
18.2 Object Management	289
18.3 Tasks	289
18.4 Critical Regions	289
18.5 Scheduling	289
18.6 I/O	290
18.7 Messaging	290
18.8 Fault Containment	290
18.9 Error Recovery	291
18.10 Overload	291
18.11 Processor Failures	292
18.12 Operability	292
18.13 Software Installation	292
18.14 Debugging	294
18.15 Summary	294
Glossary	295
References	309
Index	315

About the Author

Greg Utas received his honors BSc in Computer Science from the University of Western Ontario (Canada). He joined Nortel Networks in 1981, where he served as the principal software architect for various switching products. As chief software architect of GSM NSS Development, he led a team of 50 designers who redesigned the product's call processing software using object-oriented techniques. For this work, he received the Nortel Technology Award for Innovation and became the first software architect at Nortel's director level. In March 2002, Greg joined Sonim Technologies as chief software architect, responsible for the design of push-to-talk services for wireless networks. He recently left Sonim to become a consultant specializing in the design of carrier-grade software.

Greg has presented papers at the International Switching Symposium, the International Workshop on Feature Interactions in Telecommunications and Software Systems, and at ChiliPLoP, a patterns conference. He has also authored a patterns paper in *IEEE Communications* and contributed a chapter to the book *Design Patterns in Communication Systems*.

Readers can e-mail Greg at greg@carriergradesoftware.com. He plans to post discussions related to the book at www.carriergradesoftware.com.

Preface

This book is about programming techniques for building carrier-grade software systems. Carrier-grade software is required in products that support mission-critical applications. These products include routers, switches, servers, and gateways found in communication networks. Firms that operate such networks are known as carriers, so the term carrier-grade refers to a product that meets their stringent quality requirements. However, the need for carrier-grade software is now emerging in other products, such as high-end web servers, storage area network equipment, and video-on-demand servers.

Carrier-grade software must meet extreme availability, reliability, and scalability requirements. Consequently, it employs many techniques that are not common practice in the computing industry. These techniques have been proven in products designed by firms such as Lucent, Nortel, and Ericsson. However, they have never been documented in a comprehensive manner. Instead, they are lore among the software engineers who develop products that depend on them. This book attempts to elucidate what is currently a black art.

TARGET AUDIENCE

This book is primarily intended as a guide for software engineers who work on products that face carrier-grade requirements. Managers and testers of such products will also find it useful. It could also serve as a reference text for advanced software engineering courses on highly available, distributed systems that support connection-oriented protocols.

This book will also help those who perform due diligence on products that claim to be, or need to become, carrier grade. This form of due diligence often neglects to assess a product's software architecture, even though this is a primary determinant of whether a product is, or can become, carrier grade. To serve this purpose, the last chapter of this book defines maturity levels for carrier-grade software. Development teams can also use these levels for self-assessment purposes.

MOTIVATION

The impetus for this book began at the 1998 ChiliPLoP conference in Arizona. At that conference, I participated in the TelePLoP hot topic, which was mostly attended by software architects working in the telecom industry. The other group members were Ward Cunningham, Dennis DeBruler, David DeLano, Jim Doble, Bob Hanmer, John Letourneau, and Greg Stymfal. Jim Coplien organized the hot topic. Linda Rising was the conference chair and therefore could not attend our sessions, but she later edited *Design Patterns in Communications Software* [RIS01], which included patterns written by TelePLoP participants. However, few of these patterns were programming patterns, and there were many carrier-grade topics which they did not cover.

We all felt that a comprehensive book on carrier-grade software techniques was needed. Nothing close to one existed. It was as if we all lived in pre-literate societies, where knowledge was handed down in an oral tradition. We could all recite cases in which teams developing new products were unfamiliar with techniques that we took for granted. In other cases, teams exposed to these techniques dismissed them in favor of others that were common practice in the computing industry and, therefore, surely better than bizarre ideas being touted by dinosaurs. Both situations led to products that failed to meet carrier-grade requirements. These products then had to be cancelled or rewritten, or their customers had to suffer until they could replace them. We believed that a book that included the rationale for our techniques could help to avert such unfortunate outcomes.

EXAMPLES

For many reasons, most of the examples in this book come from the telecom field. It is the most mature domain when it comes to experience in constructing carrier-grade software. Conveniently, readers

also have a general familiarity with the domain, given that everyone makes telephone calls. It is also an area in which I have worked as a software architect for over twenty years, so it offers me a ready source of examples that I know to be realistic rather than hypothetical.

The book contains a number of code examples. They are intended as sketches to illustrate the concepts described in the text. They are *not* carrier-grade in every detail, but they *do* convey the essence of what has been used in carrier-grade systems.

Typefaces are used as follows:

- Bold face (e.g. **five nines**) introduces technical terms.
- Small caps (e.g. OBJECT POOL) highlight names of techniques and patterns.
- COURIER (e.g. `vptr`) is used for code examples and types.
- COURIER ITALICS (e.g. *setitimer*) denote operating system calls.

Acknowledgments

For the past ten years, I worked for Subramaniam ‘Loga’ Logarajah in his role as VP of Engineering for various products that faced carrier-grade requirements. It is rare to find people in this type of role who demonstrate Loga’s degree of commitment to software architecture. I wish to thank Loga for his support and trust, which allowed me the opportunity to acquire the experience needed to write this book.

After completing the first draft of this book, I received many thoughtful comments from reviewers. These ran to almost fifty pages, encompassing both style and technical comments that significantly improved the final draft. For their helpful and timely feedback, I wish to thank Dennis DeBruler, Jim Doble, Steven Fraser, Bob Hanmer, Stephanie Ho, Kim Holmes, Chris Honkakangas, Jagdish Patel, and Linda Rising.

Carrier-Grade Software Techniques

Purposes

- [a] availability
- [c] capacity
- [p] productivity
- [r] reliability
- [s] scalability

A → B A is a prerequisite for B
Technique use judiciously

