UNIVERSITY OF CALIFORNIA,
IRVINE

Extending the REpresentational State Transfer (REST) Architectural Style for Decentralized Systems

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Rohit Khare

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Debra J. Richardson
Professor André van der Hoek

2003

Dedication

To my family,

For the love and encouragement only
many, many generations of peer pressure
could possibly provide!

Sri Uma Shankar Srivastava
Sri Dwarika Nath Srivastava
Dr. Mohan Khare
Dr. Bishun Khare
Dr. Jyoti Khare
Dr. Ram Srivastav

Dr. Reena Khare
Dr. Smruti Vidwans
~~*Dr.*~~ Adam Rifkin, A.B.D.

&

Dr. Mom !!!

I can only hope this volume serves half as well to

inspire the next generation of scholars in our family...

(to be continued...)

Table of Contents

Page

List of Tables ix

List of Programs x

List of Figures xi

Acknowledgments xiii

Curriculum Vitae xv

Abstract of the Dissertation xxi

Chapter 1: Introduction 1

1.1 Scenario 1

1.2 Approach 4

1.3 Contributions 5

1.4 Validation 6

1.5 Organization 6

Chapter 2: Problem Analysis 9

2.1 Factors 9

2.1.1 Latency 10

2.1.1.1 Physical Limits 11

2.1.2 Agency 13

2.2 Formal Model 13

2.2.1 Simultaneous Agreement 14

2.2.2 Related Models 16

2.3 Definitions 19

2.3.1 Centralized 20

2.3.2 Distributed 21

2.3.2.1 Delegated 24

2.3.3 Estimated 25

2.3.4 Decentralized 27

2.3.5 Resource 29

2.3.6 Representation 30

2.4 Problem Statement 31

CHAPTER 3: ARCHITECTURAL APPROACHES TO DECENTRALIZATION 33

3.1 SOFTWARE ARCHITECTURE 33
3.1.1 Example: C2 35
3.2 MESSAGE-ORIENTED MIDDLEWARE 36
3.3 INTERNETWORKING 37

CHAPTER 4: UNDERSTANDING REST 39

4.1 MODERN WEB ARCHITECTURE 39
4.1.1 An Architectural Style for the Web 40
4.1.2 Limitations of the "One-Way Web" 42
4.2 REPRESENTATIONAL STATE TRANSFER (REST) 44
4.2.1 Property Specification 44
4.2.2 Style Definition 47
4.2.2.1 The `GlobalClock` Component 48
4.2.3 Validation 48
4.2.4 Implementation Issues 49
4.2.4.1 Expiration 49
4.2.4.2 Synchronization 50
4.3 REST WITH POLLING (REST+P) 50
4.3.1 Property Specification 51
4.3.2 Style Definition 53
4.3.2.1 The `PollingClient` Component 54
4.3.3 Validation 54
4.3.4 Implementation Issues 54

CHAPTER 5: CENTRALIZED SYSTEMS 57

5.1 ASYNCHRONOUS REST (A+REST) 57
5.1.1 Property Specification 57
5.1.2 Style Definition 60
5.1.2.1 The `NotifyingOriginServer` Component 61
5.1.3 Validation 62
5.1.4 Implementation Issues 62
5.2 ROUTED REST (R+REST) 63
5.2.1 Property Specification 64
5.2.2 Style Definition 65
5.2.2.1 The `RoutingProxy` Component 68
5.2.3 Validation 69
5.2.4 Implementation Issues 69

5.3 Asynchronous, Routed REST (ARREST) 70
5.3.1 Property Specification 70
5.3.2 Style Definition 72
5.3.2.1 The `CentralizedEventRouter` Component 73
5.3.3 Validation 73
5.3.4 Implementation Issues 74
5.3.4.1 Reflection 74
5.3.4.2 Scalability 75
Chapter 6: Distributed Systems 77
6.1 Property Specification 77
6.1.1 Atomic 78
6.1.2 Consistent 79
6.1.3 Isolated 79
6.1.4 Durable 80
6.2 REST with Delegation Decisions (REST+D) 81
6.2.1 Style Definition 81
6.2.1.1 The `MutexLock` Component 82
6.2.2 Validation 83
6.2.3 Implementation Issues 84
6.3 Asynchronous, Routed REST with Distributed Decisions (ARREST+D) 85
6.3.1 Style Definition 86
6.3.1.1 The `FairMutexLock` Component 87
6.3.2 Validation 88
6.3.3 Implementation Issues 89
Chapter 7: Consensus-Free Systems 91
7.1 The 'Now Horizon' 91
7.2 BASE Representation Transfers 94
7.2.1 Best-Effort 96
7.2.2 Approximate 96
7.2.3 Self-Centered 97
7.2.4 Efficient 97
7.3 Risk Management 98
Chapter 8: Estimated Systems 101
8.1 Property Specification 101
8.2 REST with Estimates (REST+E) 102

8.2.1 Style Definition 102
8.2.1.1 The TCP Connector 104
8.2.1.2 The Cache Connector 104
8.2.1.3 The AccessControl Connector 105
8.2.1.4 The ContentNegotiation Connector 105
8.2.2 Validation 106
8.2.3 Implementation Issues 107
8.3 Asynchronous, Routed REST with Estimates (ARREST+E) 107
8.3.1 Style Definition 107
8.3.1.1 The StoreAndForward Connector 108
8.3.1.2 The Summarizer Connnector 109
8.3.1.3 The Predictor Connector 109
8.3.1.4 The TrustManager Connector 110
8.3.2 Validation 111
8.3.3 Implementation Issues 112
8.3.3.1 Retransmission Strategy 112
8.3.3.2 Impedance Matching 113
8.3.3.3 Stateful Predictors 114
8.3.3.4 Credential Management 114
Chapter 9: Decentralized Systems 115
9.1 Asynchronous, Routed REST with Estimates and Decentralized Decisions (ARRESTED) 115
9.1.1 Property Specification 116
9.1.1.1 Assessment 116
9.1.1.2 Correspondence 117
9.1.2 Style Definition 117
9.1.2.1 The Assessor Component 118
9.1.3 Validation 119
9.1.4 Implementation Issues 120
Chapter 10: Infrastructure 123
10.1 The Mod_PubSub Project 123
10.1.1 Approach: Application-Layer InterNetworking (ALIN) 124
10.1.2 Design 125
10.1.2.1 Protocol 125
10.1.2.2 Event Model 126
10.1.2.3 Router 127
10.1.2.4 Microserver 128
10.1.2.5 Applications 129

10.1.3 Implementations 131
10.1.3.1 Open Source *132*
10.1.3.2 Proprietary *135*
10.2 Feasibility of Our New Styles 136
10.2.1 Existing Elements 137
10.2.1.1 Synchronization (REST) *137*
10.2.1.2 Polling (REST+P) *137*
10.2.2 New Elements 138
10.2.2.1 Asynchrony (A+REST) *138*
10.2.2.2 Routing (R+REST) *138*
10.2.2.3 Delegated Decisions (REST+D) *139*
10.2.2.4 Estimation (REST+E) *140*
10.2.3 Composite Elements 140
10.2.3.1 Event Routing (ARREST) *140*
10.2.3.2 Distributed Decisions (ARREST+D) *141*
10.2.3.3 Prediction (ARREST+E) *142*
10.2.3.4 Decentralized Decisions (ARRESTED) *143*
10.3 Evaluation 144
10.3.1 Scalability 145
10.3.1.1 Clustering *146*
10.3.2 Applicability 148
10.3.2.1 Journaling *149*
Chapter 11: Applications 151
11.1 Auction Markets 151
11.2 Methodology 154
11.3 AutoMarket 156
11.3.1 Centralized Auction 157
11.3.2 Distributed Auction 159
11.3.3 Estimated Auction 159
11.3.4 Decentralized Auction 160
11.4 Evaluation 161
Chapter 12: Conclusions 163
12.1 Summary 163
12.1.1 Problem Statement 163
12.1.2 Problem Analysis 163
12.1.3 Insight 164

12.1.4 Approach 165
12.1.4.1 New Centralized Styles *167*
12.1.4.2 New Distributed Styles *169*
12.1.4.3 New Estimated Styles *169*
12.1.4.4 New Decentralized Style *171*
12.1.5 Evaluation 171
12.2 CONTRIBUTIONS 172
12.3 FUTURE WORK 173
REFERENCES 177

List of Tables

Page

Table 1: Classifying types of variables by control and consensus requirements. . 19

Table 2: Rules for selecting valid representations for each type of resource. 31

Table 3: Summary of the REST style. 48

Table 5: Summary of the REST+P style. 54

Table 6: Summary of the A+REST style. 62

Table 7: Summary of the R+REST style. 69

Table 8: Summary of the ARREST style. 74

Table 9: Summary of the REST+D style. 83

Table 10: Summary of the ARREST+D style. 88

Table 12: Summary of BASE constraints and mechanisms. 95

Table 13: Summary of the REST+E architectural style. 106

Table 14: Summary of the ARREST+E architectural style. 111

Table 15: Summary of the ARRESTED architectural style. 119

Table 16: An inventory of files in the `Mod_PubSub` project. 133

Table 17: Summary of our new architectural elements and their implementations. 145

Table 18: Properties of five major categories of event-based applications. 149

Table 19: Summary of observations from implementing `AutoMarket` in each style. 162

Table 20: Summary of our centralized architectural styles. 168

Table 21: Summary of our distributed architectural styles. 169

Table 22: Summary of our estimated architectural styles. 170

Table 23: Summary of our decentralized architectural style. 171

Table 24: Six new properties and the styles constructed to induce each. 172

List of Programs

Page

Program 1: Specification of *read()*. 45

Program 2: *read()* quantified over possible latencies. 45

Program 3: *read()*, if the variable has already been assigned a value. 46

Program 4: Specification of *fresh()*. 46

Program 5: Composition of *fresh()* and *read()*. 47

Program 6: Specification of *poll()*. 51

Program 7: Repeating *poll()* when values expire. 52

Program 8: Specification of *wait()* 53

Program 9: Specification of *notify()* 57

Program 10: *Notify()*, quantified over possible latencies. 58

Program 11: Composition of *fresh()* and *notify()* 58

Program 12: Specification of *subscribe()*. 58

Program 13: Nested composition requires transitive trust. 64

Program 14: Sequential composition only requires direct trust. 65

Program 15: Specification of *delegate()*. 65

Program 16: Nested composition of *delegate()* only requires direct trust. 65

Program 17: Typical presentation of Lamport's Bakery algorithm for process *i*. 87

Program 18: Simplified Bakery algorithm using a clock and a network. 88

Program 19: Pseudocode for implementing a `FairMutexLock` peer. 90

Program 20: A sample JavaScript `Mod_PubSub` chat application. 135

Program 21: Format of a `bid` event in `AutoMarket`. 158

List of Figures

Page

Figure 1: Latency induces uncertainty for traders "further" away from a centralized resource. 10

Figure 2: The shaded region illustrates an interval of simultaneous agreement. 15

Figure 3: The Representational State Transfer Style, from [67] 41

Figure 4: Illustration of the REST architectural style. 47

Figure 5: Illustration of the REST+P architectural style. 53

Figure 6: Illustrating simultaneous agreement using notifications with leases. 59

Figure 7: Illustration of the A+REST architectural style. 61

Figure 8: Multilateral extensibility requires eliminating untrusted links. 66

Figure 9: Illustration of the R+REST architectural style. 67

Figure 10: World-lines illustrating the total latencies for simultaneous invocation in each of our centralized styles. 71

Figure 11: Illustration of the ARREST architectural style. 72

Figure 12: Illustration of the REST+D architectural style. 82

Figure 13: Illustrated example of a 3-way shared resource in the ARREST+D architectural style. 86

Figure 14: Logical and physical views of a transcontinental network. 92

Figure 15: Latency maps of the same transcontinental network. 92

Figure 16: An illustration of precision and accuracy using a dartboard. 95

Figure 17: Illustration of the REST+E architectural style. 103

Figure 18: Illustration of the ARREST+E architectural style. 108

Figure 19: Illustrated example of a concept shared by 5 agencies in the ARRESTED architectural style. 118

Figure 20: Components of `mod_pubsub` operate within several other systems. 124

FIGURE 21: A LISTING OF SAMPLE APPLICATIONS IN THE MOD_PUBSUB DISTRIBUTION. 129

FIGURE 22: THE INTROSPECT APPLICATION EXAMINING A FOR-SALE EVENT NOTIFICATION. 130

FIGURE 23: A MULTI-PROTOCOL EVENT ROUTER CAN INTERCONNECT MULTIPLE ORGANIZATIONS. 131

FIGURE 24: A BLOCK DIAGRAM OF THE INTERNAL ARCHITECTURE OF THE COMMERCIAL ROUTER 136

FIGURE 25: THE ZACK ACKNOWLEDGMENT-COUNTING SAMPLE APPLICATION. 141

FIGURE 26: SCREEN SHOTS OF AN EXAMPLE USED-CAR MARKETPLACE. 151

FIGURE 27: ILLUSTRATION OF A CENTRALIZED (EXCHANGE) MARKET. 152

FIGURE 28: ILLUSTRATION OF A DISTRIBUTED (BROKERED) MARKET. 152

FIGURE 29: ILLUSTRATION OF A DECENTRALIZED (OVER-THE-COUNTER) MARKET. 153

FIGURE 30: DIAGRAM SUMMARIZING DERIVATION OF OUR FOUR NEW ARCHITECTURAL STYLES 165

FIGURE 31: CONSENSUS-BASED STYLES ONLY WORK WHEN THE ENTIRE APPLICATION IS INSIDE OF THE NOW HORIZON. 166

FIGURE 32: MASTER/SLAVE STYLES ONLY WORK FOR APPLICATIONS CONTROLLED BY ONE AGENCY. 166

Acknowledgments

"Don't worry, Rohit... If you make it past AMa95, it will be downhill all the way to your PhD!" —Prof. Alain J. Martin, Caltech, 1994

If nothing else, this hefty volume must at least testify to what an incredibly difficult and frustrating experience junior year Applied Mathematics and Complex Analysis can be. Of course, the joy of Caltech is knowing that your roommate breezed right through the same course back in freshman year... sigh!

Embedded within that irony, however, is the principle that has guided me so well, so far: *surround yourself with people even better than you are*. I may be a big guy, but that's no reason to become the 'big fish' — that only means it's time to move on to a bigger pond!

Thankfully, I've had the opportunity to work with a great many people who are far more talented, dedicated, and experienced than myself at every step of the way. I know I can be quite a handful at times, but many of my mentors have been generous enough with their time and trust to permit me to pursue my own paths.

Foremost among them is, not surprisingly, my advisor and committee chair, Dick Taylor. If he weren't so supportive of all my adventures within the department — from helping organize workshops to racking up all those Incompletes! — as well as out in the 'real world,' I might not have even gotten up the courage to return to school and finish my Ph.D. Credit for that decision is also due to Debra Richardson, who as department chair (and later as interim Dean) encouraged my re-entry.

Of course, things do change when you go away for a few years (in my case, to start a reasonably successful software company). Fellow students come and go (though I know a few who ought to be going soon — good luck, Peter and Joe!), and so do the faculty. David Rosenblum, who was an essential catalyst for my own interest in the area of real-time event notification, had moved on to his own startup, while André van der Hoek had just moved in from Boulder. Working with Andre provided a fresh new perspective, helping add depth and rigor to some notions that were initially pretty far afield from what I thought I had dashed back to Irvine to do: "What I did on my (very long!) summer vacation."

Instead, this dissertation slowly but surely shifted from a systems thesis — a clever new way to view event notification as an application-layer routing problem — into a much more reflective study of software architecture. For this, I owe a unique debt to Roy Fielding, whose investigation into Web architecture laid the foundation for my own. It's rare to see a dissertation that constitutes a 'sequel' to an earlier graduate's work. This should only be read as a testament to the power of Roy's idea. I am similarly impressed by the scope and passion of the "REST advocacy" community this gave rise to, including sparring partners like Mark Baker.

Roy, in turn, was building on work of another of my mentors, Tim Berners-Lee.

Tim, and the rest of the crew at the nascent World Wide Web Consortium (W3C) — Henrik Frystyk Nielsen, Dan Connolly, Dave Raggett, Sally Khudairi, and many more — were great examples of *both* individual initiative *and* teamwork. It's hard to imagine even attempting to measure one's own work up against such benchmarks as the Web or XML, but they instilled in me the confidence to try.

What I wouldn't have predicted, though, was that some of W3C's summer interns would eventually lead me to follow them back to UC Irvine, of all places. Jim White head, who visited in the summer of 1996, pulled out all the stops to convince me to join Greg Bolcer and the rest of the HyperWare team here.

Even so, I didn't cut a straight path back to California. John Klensin, who helped guide me through some of the rockier shoals of standards politics at the IETF, tempted me to come work with Vint Cerf's Internet Architecture group at MCI. If anything, you could even say TimBL's work was laid over the foundation of TCP/IP — but it's even more chilling to assess one's works against *that* yardstick!

Instead, the day-to-day slog of academic research is a battle won or lost with ones' buddies in the trenches, not by the generals in the history books. Consider my debt of gratitude to Eric Dashofy, for everything from existential critiques of why my research should even be funded, to bailing me out of tight spots with the Registrar while I was living off-campus... *500 miles* off-campus!

However, the closest companion of mine through all these years was (and is!) Adam Rifkin. Sure, he happened to be a grad student in some *other* Southern Californian CS department, but we worked together as closely as I could have hoped these past dozen years. I literally would not be here without him, not just in publication count, but even to have the courage to actually start a company with just us, Peyman Oreizy, and a crazy idea about Instant Messaging over the Web.

.... I suppose it's always around this point that an author begins to feel overwhelmed by onrushing memories of all one's colleagues, friends, and loved ones who have shaped every word of their works. It is, after all, a fact seldom acknowledged that the only people who read acknowledgment sections are all the people who rightfully expect to see themselves in it! Please accept my apologies, every one of you, from the third-grade teacher who taught me to love reading (Mrs. Hafen) to the high-school teachers who taught me to love writing (Kathy Baer and Lynda Mitic) to the college buddies who taught me something about love itself (Ernie Prabhakar)... each and every one of the FoRKs out there, thank you!!

And last, as they say, but not least, I'd like to thank the muse who actually got me through writing this thesis. After five solid years of procrastination, I can't help but notice that I finally started cranking out chapters only after I met this certain someone — the lady who I dearly hope will soon my wife, Smruti Jayant Vidwans!

So, yes, buried here in the fine print, is my true thesis: *will you marry me, love?*

This material is based in part upon work supported by the National Science Foundation under Grant #0205724 and the UC Irvine Chancellor's Fellowship.

Curriculum Vitae

Rohit Khare

Education

2003 **Doctor of Philosophy**

University of California, Irvine
Information and Computer Science
Institute of Software Research
Advisor: Dr. Richard N. Taylor
Dissertation: *Extending the REpresentational State Transfer (REST) Architectural Style for Decentralized Systems*

2000 **Master of Science**

University of California, Irvine
Information and Computer Science
Major Emphasis: Software

1995 **Bachelor of Science with Honors**

California Institute of Technology
Major: Engineering & Applied Science
Major: Economics

Professional Experience

2000–02 *Founder & Chief Technology Officer*, KnowNow, Inc., Sunnyvale, CA

1997–99 *Research Assistant* (Chancellor's Fellowship),
School of Information and Computer Science, UC Irvine, California

1997–99 *Columnist*, IEEE Internet Computing, Los Alamitos, California

1997 *Internet Architect*, MCI, Boston, Massachusetts

1996– *President*, 4K Associates, Irvine, California

1996–98 *Editor-in-Chief*, World Wide Web Journal,
O'Reilly & Associates, Sebastopol, California

1995–97 *Member of Technical Staff*, World Wide Web Consortium,
Massachusetts Institute of Technology, Cambridge, Massachusetts

1991–95 *Research Assistant*, Undergraduate Research Fellowships (SURF),
California Institute of Technology, Pasadena, California

1989– *Programmer*, Envirosystems, Inc., Columbia, Maryland

Refereed Papers

[1] Khare, R., Guntersdorfer, M., Oreizy, P., Medvidovic, N. and Taylor, R. N. *xADL: Enabling Architecture-Centric Tool Integration with XML*, in *Hawaii International Conference on System Sciences (HICSS-34), Software mini-track*, (Maui, Hawaii, January 3-6 2001).

[2] Khare, R. and Rifkin, A. *Composing Active Proxies to Extend the Web*, in *OMG-DARPA-MCC Workshop on Compositional Software Architectures*, (Monterey, CA, Jan. 1998), ACM SIGSOFT Software Engineering Notes 23(3), pp.44-63.

[3] Khare, R. and Rifkin, A. *The Origin of (Document) Species* in *Computer Networks and ISDN Systems*, 1998, *30*. pp. 389-397.

[4] Rifkin, A. and Khare, R. *eText: An Interactive Environment for Learning Parallel Programming*, in *Proceedings of the 25th SIGCSE Symposium on Computer Science Education*, (Phoenix, Arizona, 1994), ACM, pp. 282-285.

Standards

[5] Lawrence, S. and Khare, R. *RFC 2817: Upgrading to TLS Within HTTP/1.1.* IETF, May 2000.

[6] Khare, R. and Rifkin, A. *Scenarios for an Internet-Scale Event Notification Service (ISENS).* IETF draft (expired), 13 August 1998.
`http://www.ics.uci.edu/~rohit/draft-khare-notification-00.txt`

[7] Frystyk, H., Connolly, D., Khare, R. and Prud'hommeaux, E. *PEP: An Extension Mechanism for HTTP.* W3C Technical Report, November 1997.

Edited Volumes

[8] Khare, R. (ed.), *The Web After Five Years (W3J vol. 1 no. 3)* O'Reilly & Associates, 1996. 218pp.

[9] Khare, R. (ed.), *Building an Industrial-Strength Web (W3J vol. 1 no. 4)* O'Reilly & Associates, 1996. 244pp.

[10] Khare, R. (ed.), *Advancing HTML: Style and Substance (W3J vol. 2 no. 1)* O'Reilly & Associates, 1997. 246pp.

[11] Khare, R. (ed.), *Scripting Languages: Automating the Web (W3J vol. 2 no. 2)* O'Reilly & Associates, 1997. 219pp.

[12] Khare, R. (ed.), *Web Security: A Matter of Trust (W3J vol. 2 no. 3)* O'Reilly & Associates, 1997.

[13] Connolly, D. and Khare, R. (eds.). *XML: Principles, Tools, and Techniques (W3J vol. 2, no. 4)* O'Reilly & Associates, 1997.

Technical Reports

[14] Khare, R. and Taylor, R. *Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems.* Institute for Software Research, University of California, Irvine, Irvine, CA, September 2003.
`http://www.isr.uci.edu/tech_reports/UCI-ISR-03-8.pdf`

[15] Khare, R. *Decentralized Software Architecture.* Institute for Software Research, University of California, Irvine, December 2002.
`http://www.isr.uci.edu/tech-reports/UCI-ISR-02-6.pdf`

[16] Medvidovic, N., Oreizy, P., Taylor, R. N., Khare, R. and Guntersdorfer, M. *An Architecture-Centered Approach to Software Environment Integration.* 2000.
`ftp://www.ics.uci.edu/pub/arch/papers/TR-UCI-ICS-00-11.pdf`

Articles

[17] Khare, R. and Rifkin, A. *Capturing the State of Distributed Systems with XML* in *World Wide Web Journal*, 1997, *2* (4). pp. 207-218.

[18] Khare, R. and Rifkin, A. *Weaving a Web of Trust* in *World Wide Web Journal*, 1997, *2* (3). pp. 77-112.

[19] Rifkin, A. and Khare, R. *XML: A Door to Automated Web Applications* in *IEEE Internet Computing*, July, 1997. vol. *1* (4), pp. 78-87.

[20] Khare, R. and Rifkin, A. *XML: A Door to Automated Web Applications (Japanese Translation)* in *Nikkei Electronics*, 1998 (706). pp. 221-232.

[21] Khare, R. *Telnet: The Mother of All (Application) Protocols* in *IEEE Internet Computing*, May/June, 1998. vol. *2* (3), pp. 88-91.

[22] Khare, R. *The Transfer Protocols* in *IEEE Internet Computing*, March/April, 1998. vol. *2* (2), pp. 80-82.

[23] Khare, R. *I Want My FTP: Bits on Demand* in *IEEE Internet Computing*, July/August, 1998. vol. *2* (4), pp. 88-91.

[24] Khare, R. *The Spec's in the Mail* in *IEEE Internet Computing*, September/October, 1998. vol. *2* (5), pp. 82-86.

[25] Khare, R. *Reflections on the Wizard of TPs...and Other Network News* in *IEEE Internet Computing*, November/December, 1998. vol. *2* (6), pp. 74-78.

[26] Khare, R. *Building a Perfect Beast: Dreams of a Grand Unified Protocol* in *IEEE Internet Computing*, March/April, 1999. vol. *3* (2), pp. 89-93.

[27] Khare, R. *Who Killed Gopher? An Extensible Murder Mystery* in *IEEE Internet Computing*, Jan/Feb, 1999. vol. *3* (1), pp. 81-84.

[28] Khare, R. *What's in a Name? Trust. (Internet-Scale Namespaces, Part II)* in *IEEE Internet Computing*, November/December, 1999. vol. *3* (6), pp. 80-84.

[29] Khare, R. *Anatomy of a URL (and Other Internet-Scale Namespaces, Part I)* in *IEEE Internet Computing*, November/December, 1999. vol. *3* (5), pp.78-81

[30] Khare, R. *W* Effect Considered Harmful* in *IEEE Internet Computing*, July/August, 1999. vol. *3* (4), pp. 89-92.

[31] Khare, R. *Can XForm Transform the Web? Transcending the Web as GUI, Part II* in *IEEE Internet Computing*, March/April, 2000. vol. *4* (2), pp. 103-106.

[32] Khare, R. *How <FORM> Functions: Transcending the Web as GUI, Part I* in *IEEE Internet Computing*, January/February, 2000. vol. *4* (1), pp. 88-90.

Manuscripts

[33] Rifkin, A. and Khare, R. *A Bibliography of Event Papers*. July 1998. http://www.ifindkarma.com/attic/isen/event-papers.html

[34] Rifkin, A. and Khare, R. *The Evolution of Internet-Scale Event Notification Services: Past, Present, and Future*. (Manuscript), 10 August 1998. http://www.ifindkarma.com/attic/isen/wacc/

Theses

[35] Khare, R. *Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems* (PhD Thesis), University of California, Irvine, Information and Computer Science, Irvine, CA, 2003.

[36] Khare, R. *Message-Oriented Middleware and the Software Engineer* (MS Thesis), University of California, Irvine, Information and Computer Science, 1999. http://www.4k-associates.com/moma.html

Software

[37] Khare, R., Rifkin, A., Sitaker, K. and Sittler, B. *mod_pubsub: an open-source event router for Apache*, 2002. http://mod-pubsub.sourceforge.net/

[38] Khare, R. *eText Engine: a Graphical Hypertext Editor for Electronic Textbooks*, Caltech Archetypes Project, Pasadena, CA, 1994-5. http://xent.com/~rohit/eTextEngineHome.htmld

[39] KnowNow. *KnowNow LiveServer / LiveBrowser / LiveSheet*, ver. 1.8, Sunnyvale, CA, 2003. http://www.KnowNow.com/

Talks

[40] Khare, R. *SOAP Routing: The Missing Link*, in *O'Reilly Emerging Technology Conference*, (Santa Clara, CA, 2002). http://conferences.oreillynet.com/presentations/et2002/khare_rohit.ppt

[41] Khare, R. *The Two-Way Web: An Interoperable Foundation for P2P*, in *O'Reilly Peer-to-Peer Conference*, (San Francisco, CA, 14 February 2001). http://conferences.oreillynet.com/cs/p2p2001/view/e_sess/1163

[42] Khare, R. *Internet-scale Namespaces*, in *The Workshop on Internet-Scale Technologies (TWIST'99)*, (Irvine, CA, 1999).

[43] Khare, R. *Using PICS Labels for Trust Management*, in *DIMACS Workshop on Trust Management in Networks*, (Rutgers, New Jersey, 1 October 1996). http://dimacs.rutgers.edu/Workshops/Management/Khare.html

[44] Khare, R. *Security Extensions for the Web*, in *RSA Data Security Conference*, (San Francisco, CA, 19 January 1996). http://w3.org/Talks/960119-RSA/

Teaching

[45] Khare, R. and Rifkin, A. *XML: The Least You Need To Know (Tutorial)*, in *Swiss Group for Object-Oriented Systems and Environments (CHOOSE)*, (Ausbildungszentrum Bankverein Basel, Switzerland, 19 January 1998).

[46] Khare, R. and Whitehead, J. *XML and WebDAV: Emerging Web Standards and Their Impact on Software Engineering (Tutorial)*, in *Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, (Lake Buena Vista, Florida, 5 November 1998).

[47] Khare, R. and Rifkin, A. *XML: Modeling Data and Metadata (Tutorial)*, in *Proceedings of the 1998 ACM Conference on Computer-Supported Cooperative Work (CSCW'98)*, (Seattle, WA, 15 November 1998). http://www.ics.uci.edu/~rohit/cscw98/

[48] Khare, R. and Rifkin, A. *XML: Modeling Data and Metadata (Tutorial)*, in *Proceedings of the International Conference on Work Activities, Coordination, and Collaboration (WACC'99)*, (San Francisco, California, 22 February 1999), pp. 239-240.

[49] Khare, R. and Whitehead, J. *WebDAV: Collaborative Web Authoring (Tutorial)*, in *9th International World Wide Web Conference*, (Amsterdam, The Netherlands, 15 May 2000).

Abstract of the Dissertation

Extending the REpresentational State Transfer (REST)
Architectural Style for Decentralized Systems

by

Rohit Khare
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 2003
Professor Richard N. Taylor, Chair

Because it takes time and trust to establish agreement, traditional consensus-based architectural styles cannot safely accommodate resources that change faster than it takes to transmit notification of that change, nor resources that must be shared across independent agencies. There are physical and logical limits that make simultaneous agreement (a strong form of consensus for read/write variables) expensive and ultimately, impossible. In practice, software architects resolve this contradiction by assuming that network latency is negligible and that computers operated by independent agencies are reliable — two increasingly shaky assumptions about integrating services across the Internet.

Our approach to this challenge is architectural: proposing constraints on the configuration of components and connectors that induce desired properties of the whole application. Specifically, we present, implement, and evaluate variations of the World Wide Web's REpresentational State Transfer (REST) architectural style that are optimized for centralized, distributed, estimated, and decentralized systems.

For centralized resources, we enforce simultaneous agreement by extending REST into an event-based architectural style by adding Asynchronous event notification and Routing through active proxies (ARREST). For distributed control of shared resources, we enforce ACID transactions by further extending REST with end-to-end Decision functions that enable each component to serialize all updates (ARREST+D).

The alternative to simultaneous agreement is *decentralization*: permitting independent agents to make their own decisions. This requires accommodating four intrinsic sources of uncertainty that arise when communicating with remote agencies: loss, congestion, delay, and disagreement. Their corresponding constraints are Best-effort data transfer, Efficient summarization of data to be sent, Approximate estimates of current values from data already received, and Self-centered trust management.

These so-called 'BASE' properties can be enforced by replacing references to shared resources with end-to-end Estimator functions. Such extensions to REST can increase *precision* of measurements of a single remote resource (ARREST+E); as well as increase *accuracy* by assessing the opinions of several different agencies (ARRESTED).

The contributions of this dissertation include: a formal definition of decentralization; an analysis of the limitations of consensus-based software architectural styles; derivation of new architectural styles that can enforce the required properties; and implementations that demonstrate the feasibility of those styles and sample applications.

Chapter 1: Introduction

We are interested in designing decentralized software for a decentralized society — systems that will permit independent citizens, communities, and corporations to maintain their own models of the world. Portions of such applications must operate under the control of multiple, independent administrative authorities (agencies); and may be physically separated to the extent that communication latency between those parts becomes a significant factor in their design.

The state of the art in software engineering has long focused on designing solutions for distributed systems, which presume consensus is always possible. Physics abandoned simultaneity with relativity in 1905; formal models of computing disproved consensus over faulty, asynchronous networks in 1985 [71]. Regardless of how dominant centralized client/server architectures may appear to be today, the physical limits of latency and the social limits of free agency will make decentralization inevitable for software as well.

Our approach to coping with uncertainty and disagreement is based on the study of software architecture: constraints on configurations of components and connectors that induce desired properties of an overall system. This dissertation introduces several new architectural styles that are expressly designed to accommodate decentralization. First, we developed a formal model of the problem that allows us to analyze the limitations of consensus-based architectural styles; and to precisely define the properties of centralized, distributed, estimated, and, ultimately, decentralized resources. Second, we addressed those limitations by proposing new architectural elements and constraints that could be added to an existing network-based architectural style to induce each of those properties. Third, we validate the feasibility of those newly derived styles by implementing the infrastructure for, and applications of, each.

1.1 Scenario

Electric utilities are an example of an industry that is embracing decentralization, and could well motivate development of novel architectural styles. Unlike physical goods, electricity cannot be stockpiled effectively; therefore, power generation and consumption have to be maintained in continuous equilibrium.

The origin of the hardware/software system for controlling electricity distribution today is essentially centralized [47]. Local, regional, and international grids are interconnected through single-agency control centers, typically with a public mandate to preserve equilibrium supplies at all costs (i.e. they are independent of any private generator or consumer).

Thus, there is a closed decision loop today between firing up an air conditioner and firing up an additional gas turbine in a faraway power plant. Each increase in demand or drop in supply voltage, respectively, trickles up the "chain of command" to those control centers.

However, the widely acknowledged vision of the future of power generation, distribution, and storage is to support a vast array of much smaller-scale devices, potentially owned and operated directly by consumers and businesses [29]. To be sure, relatively few customers today can operate their own closed-cycle cogeneration plants, or sell solar power back into the grid — in part, because both joining and leaving the public grid can be quite complex, legally and financially [40].

According to this vision, there won't be a convenient readout on the big board stating exactly how many watts are being consumed moment-by-moment; indeed, due to network latency and privacy regulations, it may not be able to speak of such a number anymore. Especially once the decision to bring capacity on-line to service that air-conditioner is made locally, say, by the fuel-cell-cum-water-heater in your neighbor's garage.

A traditional client/server solution for developing a control system for distributed energy resources (DER, [108, 173]) would be to place sensors at each producing or consuming node. Those meters would be assumed to be trusted, because they are property of the utility. Similarly, telemetry streams would be trusted insofar as the readouts could be faulty, but not maliciously manipulated. Also, the telemetry data would be presumed to arrive at the control center in real–time.[1] In this setting, solving for equilibrium is a simple matter of toting up all the rows in a database table and thence deciding whether to activate additional generating capacity, or to initiate rolling blackouts.

By contrast, a truly decentralized future for power generation must permit new generation, storage, and distribution elements to be independently owned, perhaps by consumers themselves. No longer would an architect presume that trustworthy sensor data could be collected and analyzed at one location in real-time. Instead, the software that controls every single energy–producing or –consuming device would be operated on behalf of a separate agency. Thus, devices could not automatically trust any information coming in from other agencies without an elaborate trust-management mechanism (such as a public key infrastructure). That could even

[1] Since the most time-sensitive operation in a control center is throwing open an interconnect (requiring phase alignment of 60Hz loads), the minimum observable unit of time is ~10msec. That is greater than the speed-of-light delay for any regional grid, so as long as the network latency from sensors to controllers is minimized (as circuit-switched telephone networks indeed do). Thus, the entire grid control system can effectively operating in real-time (i.e. effectively zero latency) and can apply consensus-based architectural styles.

include forged bids and trades, usage levels, fuel supply levels, or even falsified weather forecasts in order to take advantage of a system without central controls. Furthermore, this would no longer be a fully-interconnected grid, in order to support new use cases such as ad-hoc battlefield power supplies, self-contained campus plants, or mobile sources like cars & trucks. All of this increases latency, not only through use of slower, public Internets, but also by adding polling delays, such as for remote devices that might only dial-in once per day.

These concerns about latency and agency pose unique risks in this context:

Price Volatility. Setting the price of electricity is a basic mechanism for aligning supply and demand — but there wouldn't be a single price point any more. Since every owner can decide whether or not to turn their equipment on or off independently, then they might as well be able to control its current price. Device control software must support such independence by replacing references to a "price" variable with estimates of the best price available from a long list of offers. This has the potential to make the ultimate price of electricity quite volatile, depending on time of day, marginal usage level, and the physical location of producers and consumers.

Arbitrage. Decentralization could create new ways to 'game the system,' because news about the "real world" may permit coalitions of providers to manipulate supplies or reschedule demand for their benefit. Private producers might not be legally obliged to keep the system running smoothly the way regulated public utilities are, and certainly can't be forced to through technological means alone.

Insecurity. An active attack is the extreme case of the risks above. Any decision to trust an external agency becomes a presumption worthy of subversion. If there were a single control message that could tell a malfunctioning device to shut down, then forging such messages could take out the entire system. If there were not a trusted settlement system, could attackers steal power without paying for it?[2]

Unreliability. Eliminating a single point of failure also eliminates a single point of control. Hierarchically delegated regulatory authority establishes an agency responsible for maintaining reliable power. If the role of regulation is relegated from active management of the grid to merely establishing standard protocols, then we may not even be able to know what systemic vulnerabilities we might face from, say, an earthquake.

[2] In trading applications, this is known as counterparty settlement risk — the risk that the buyer may become insolvent before the trade actually clears. Perhaps the most famous example of such havoc is "Herstatt risk", named for a German bank that collapsed in the middle of a workday on June 26,1974 — it took several years to sort out which overseas correspondent banks would get their money back after the German government decreed that domestic obligations were to be given priority [2, 52].

Each of the risks enumerated above are also opportunities. After all, the cardinal virtue of decentralization is that it even though it may increase risk, it should also reduce our exposure. Unlike a total regional failure, the entire system should still keep working to *some* degree after a disaster. Society will also benefit from more efficient allocation of resources overall: it may well be 'better' for some neighborhoods to pay much more than others rather than artificially subsidize each other.

Control software for this new world not only needs to establish interoperability standards [55] — what kind of power will I get? For how long? At what price? — but also needs to actively establish approximate equilibrium based on *local* information alone. Device control software might even grow complex enough to simulate speculative futures markets based on estimates of when a battery may discharge, whether an accident knocks out a transformer, or increased demand for a cogenerated byproduct, such as hot water. Moreover, that doesn't even begin to incorporate the emergent, serendipitous benefits of integrating energy management with other information systems: What if your house learned your return flight was cancelled and avoided heating the pool to begin with?

What we *are* fairly certain of, though, is that current architectural styles are poorly matched to these challenges. Software applications for decentralized "power webs" may well resemble efforts to decentralize "computing power" today: grid computing or so-called web services are just two approaches for advertising, discovery, invocation, and performance monitoring of the applications that will run within power devices. Applications to control power webs will require robust trust management, to filter out intruders who join the web with the intent of disrupting it. They may also require sophisticated statistical modeling to optimize entire power supply chains under the influence of uncertainties due to network latency and independent agency. Based on our newly developed styles, we will revisit this scenario in §12.3 with our own recommendations for developing such software.

1.2 Approach

Our approach to the challenge of developing software for decentralized systems is *architectural:* proposing constraints on the configuration of components and connectors that induce desired properties of the whole application. Since the initial context of our interest in decentralization was Internet-scale application integration, we chose to begin with REpresentational State Transfer (REST), the architectural style behind one of the most popular decentralized applications to date: the World Wide Web.

REST was originally developed to explain the design of the Web's network proto cols and client, server, and proxy software. Its key insight was a separation of concerns between abstract resources, concrete representations, and their names that neatly decoupled the traditional requirement for stateful interaction in client/server

systems and emphasized the genericity of service interfaces. Furthermore, it accommodates multiple-agency namespaces, an essential step towards decentralization.

However, some of REST's essential features disqualify it for decentralized applications. One-way, one-shot, and one-to-one representation transfers prohibit REST servers from continuing to update clients as the representation of a resource changes over time. Nor can it connect graphs of components without forcing complete mutual trust, since its only mechanism for composition is the linear proxy pipeline.

Nonetheless, it is an appropriate starting point because the resource/ representation abstraction readily maps on to the event source/event notification abstraction required to maintain simultaneous agreement. Furthermore, robust software infrastructure already exists for REST, so we believe that our incremental stylistic extensions should correspond to incremental implementation effort for prototyping.

The approach that guides our infrastructure implementation is to follow prece dents from the design of a successful infrastructure that already exhibits these properties: the Internet. The dominant model for software integration today is to use binary components with Application Programming Interfaces (APIs). We believe there is a new model for packaging software as network-accessible services using open, standard application-layer messaging protocol interfaces. We call this approach Application-Layer Internetworking (ALIN).

The approach that guides our application implementation is to investigate a domain that directly reflects the range of centralized, distributed, estimated, and decentralized social structures: auction markets. While there is a wide variety of examples beyond merely focusing on prices (such as simulation or calendaring, to name two), establishing market equilibrium is still the archetypal decentralized problem.

1.3 CONTRIBUTIONS

First, we specify a formal model and definitions of the heretofore ill-defined terms centralized, distributed, estimated, and decentralized, as well as testable properties of each type of variable and resource.

Second, we design a family of architectural styles derived from REST that induce several properties that REST alone could not: simultaneous agreement, ACID simultaneous agreement, BASE approximate agreement, and complete consensus-freedom.

Third, we develop open-source implementations for these styles, including multi-protocol, application-layer event routers, management tools, and sample applications.

1.4 Validation

Architectural styles for software are often descriptive efforts, but our hypothetical styles are prescriptive instead. The study of software architecture has typically been based on historical accumulation of experience, such as in the design patterns community [86]. REST was articulated some years after the World Wide Web was already successful. Even in this investigation, most of our ideas come from reflecting upon the construction of several practical event routers and building applications with them, rather than theory alone.

The standard method of evaluating an architectural style is observation in practice: analysis of existing systems [203], and even controlled experiments where the same application is built in multiple styles [92, 211]. But if we may term that the *inductive* approach to identifying styles, by the properties they induce from examples, then this dissertation takes a *deductive* approach, because it prescribes specific new stylistic constraints that induce desired properties. The appropriate measure of a deductive solution is to formally demonstrate that it is correct by construction.

We will also illustrate that the resulting styles are indeed feasible to implement and to apply. While we could make our points using some of the wide range of application samples included in the `Mod_PubSub` project, we will introduce a coherent family of auction applications that span the range of centralized, distributed, estimated, and decentralized marketplaces.

Specifically, we adapt a used-car marketplace to support: centralized control of sale prices; distributed control using a best-price auction; estimated control in a GUI interface that continues to display a price range once disconnected from the network; and decentralized control of a generic concept such as "truck prices," assessed from a series of individual vehicle auctions.

1.5 Organization

The basic organizational pattern of this dissertation is the pairing of a desired property and the architectural constraints that induce it. We design several new styles using this pattern of alternating subsections: specification of the required properties, a definition of the new style, validation that the new style correctly implements the abstract specification, and implementation issues encountered in practice.

We begin with unmodified REST (Chapter 4) and proceed to derive a family of consensus-based styles for centralized (Chapter 5) and distributed resources (Chapter 6). At that juncture, however, we encounter the limits of consensus and the necessity of decentralization. Developing applications on networks with excessive latency, or that span agency boundaries, requires splitting a resource into many local, independent resources. Chapter 7 introduces our general insight for proceeding outside the so-

called ‘now horizon’: explicitly managing the risk of disagreement. This will inform our development of specific new properties and new styles for both estimated (Chapter 8) and decentralized resources (Chapter 9).

The remainder of the dissertation focuses on implementing infrastructure for, and sample applications of, each of these new styles. First, Chapter 10 discusses our approach towards connecting decentralized components, Application-Layer Internetworking (ALIN); the design of our application-layer Transfer Protocol (TP) router; several open-source and commercial implementations of the same; and concludes by identifying and comparing aspects of our solution that reflect aspects of each new style. Second, Chapter 11 illustrates the feasibility of our approach towards assembling decentralized components by presenting a simple methodology and applying it specifically to an archetypal auction market application.

Finally, Chapter 12 summarizes our work and identifies our specific contributions, as well as outlining an agenda for future research in this area.

Chapter 2: Problem Analysis

Like any other design discipline, software development is subject to the vagaries of fads and fashion. In recent years, there has been a surge in the popularity of the term 'decentralization': we hear of 'decentralized file-sharing,' 'decentralized supercomputers,' 'decentralized namespaces,' and a slew of similar claims of 'peer-to-peer,' 'Internet-scale,' and 'service-oriented' architectures [59, 155, 176].

To date, the software engineering and software architecture literature has not embraced a formal definition of 'decentralization.' Indeed, in a full-text search of the ACM Digital Library, we found that it was often considered a synonym for 'distribution' until only recently. Even as of 1998, it only occurs in the official ACM subject classification once [14], and then only with respect to the organization of MIS departments and users [94, 127].

Our goal is to provide precise, testable definitions for each of our key design regimes using well-known formal models of distributed computing. To that end, this chapter will proceed to discuss the factors leading to decentralization, provide a formal definition of simultaneity in terms of the consensus problem, and use that, in turn, to formally define the properties of centralized, distributed, estimated, and decentralized resources. All these are prerequisites for stating the specific problem we intend to investigate: deriving new architectural styles that can assist in the development of applications that use such resources.

2.1 Factors

Let us consider an e-commerce example for elucidating the factors leading to decentralization. In an ideal world, if a bookseller's website says they still have one copy of a novel left for sale, there would really be one left. In practice, there may not.[3] It may have been sold by the time you hit the 'buy' button. Or, the seller may just be lying about its scarcity to drive up the price. Today's applications have few safeguards to prevent processing out-of-date or untrusted data. These errors, though seemingly minor today, will compound into severe commercial and security risks once we routinely expect to chain together services provided by multiple organizations.

The essence of decentralization is that there can be more than one answer to the same question. In this case, the question "How many copies are for sale?" has one answer according to the seller's software, another according to the (properly skeptical) buyer's software. Furthermore, even *precise* knowledge of the bookseller's claimed inventory at the warehouse may not be *accurate*. After all, it could be burning down

[3] "In theory, theory and practice are the same. In practice, they differ."

— Jan L. A. van de Snepscheut [223]

to the ground at that very moment, no matter what the table in the "reliable" inventory database server might say.

Informally, we have identified two basic factors that can force systems to accommodate more than one answer: it may take too long to find out that one answer, or it may not be legally possible to mandate one answer. The former is an absolute physical limit: latency; the latter is an absolute social limit: agency. Later, we will argue that these two forces correspond to the two conditions that make consensus formally impossible (asynchronous networking and faulty processes, §2.2). For the moment, though, the next two subsections will discuss the causes and effects of latency & agency.

2.1.1 Latency

It is important to understand the sources of latency because latency constrains the physically realizable configurations of shared state in a software architecture, or conversely, for understanding the limits of an application running on a given set of machines. Money can buy exponentially increasing computing, communications, and storage capacity, but latency is absolute.

Latency makes simultaneous agreement impossible in many real-world situations. Consider how it affects a stock traded on the (centralized) New York Stock Exchange. A stockbroker in London interested in knowing the current price of the stock consults a server that broadcasts its current price. Because of fundamental physical limits like the speed of light through fiber optic cable, she can only know what the price of stock was several milliseconds ago, at best. Realistically, Internet delays could range up to two seconds, or worse. Thus, it is impossible for the London stockbroker to know the *current* price of a heavily-traded stock traded in New York, as shown in Figure 1.

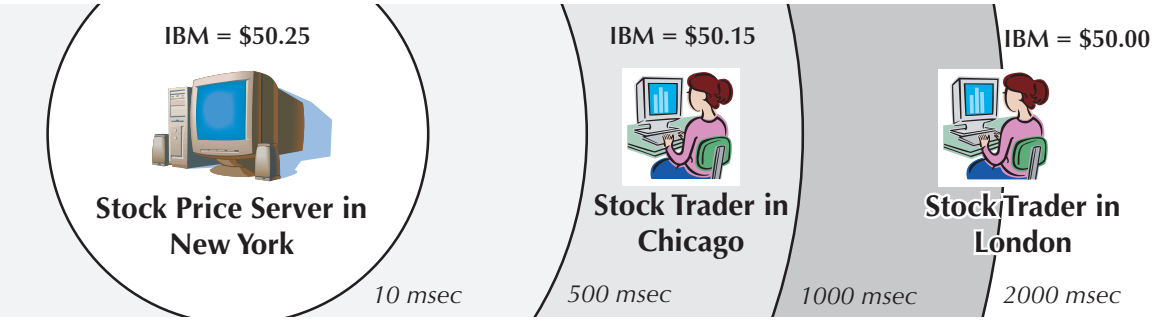

Figure 1: Latency induces uncertainty for traders "further" away from a centralized resource.

This is not an untenable situation, provided the stock price holds steady for longer than two seconds. If the price of the stock changes every ten seconds, for instance, then the London stockbroker has a full six seconds to place a buy order and

send her request before the price of the stock changes again (remember, we presume her order will take up to another two seconds to propagate back to the server in New York). The problem occurs when the interval between changes in price drops *below* two seconds. In this case, the London stockbroker's information is *always* stale. Even at four seconds between updates, it arrives too late to act upon reliably. (Furthermore, on many new networks, the maximum delay is much longer than two seconds: consider the case of wireless carriers buffering traffic while the caller drives through a tunnel.)

The concentric circles in Figure 1 represent latencies using a logarithmic scale. Their radii are correlated with distance, but more intriguingly, also determine the maximum update rate of an event source. In §7.1, we will term this a 'now horizon', since it demarcates which components can reliably refer to the value of a variable 'right now.'

While high-frequency data have small now horizons, even rarely-changing data may constrain systems in the same way. Network latency also determines the *minimum reaction time* of a system.

For example, the Federal Reserve Bank overnight interest rate may stay fixed for months, but fortunes are made and lost in mere seconds after a change is announced. Waiting for the next day's newspaper coverage could prove ruinous; even such a low-frequency event source requires high-frequency observation to minimize the delay between observation and notification. Similarly, DNS is only designed for daily updates, making it impossible to create a new second-level domain name in less than 24 hours that works throughout the Internet.

2.1.1.1 Physical Limits

Feynman once observed that "When I talk about everything in the world that is happening 'now,' that does not mean anything... We cannot agree on what 'now' means at a distance." [65]

He was referring to a revolution in physics nearly a century old. Einstein contradicted Newtonian physics by positing that uniform motion in a straight line is relative — that no experiment can reveal which of two observers is moving and which is at rest. The most famous consequence of this principle is that nothing can travel faster than c, the speed of light in a vacuum.

> Interestingly, physical information is apparently, like energy, a *localized* phenomenon. That is, it has a definite location in space, associated with the location of the subsystem whose state is in question. Even information *about* a distant object can be seen as just information in the state of a local object (*e.g.* a memory cell) whose state happens to have become correlated with the state of the distant object through a chain of interactions. Information can

be viewed as always flowing locally through space, even in quantum systems [76].

Frank's discussion of the ultimate physical limits to computation, storage, and communication is the key to understanding why latency is an absolute constraint for software architects: it takes time and energy to transmit information across a channel, if one even exists.

Propagation. An alternative interpretation of relativity is causal: c is the maximum speed of bits. Popular reports of research in quantum computing using entangled particles for "quantum teleportation" and "quantum cryptography" may initially seem eerily "telepathic" (as even Einstein described the instantaneous effects of entanglement), but actual transmission of information still requires a corresponding classical message to be sent — below c — in order to recover the state [19].

Furthermore, practical communication channels actually only achieve small fractions of c: photons travel at 67% c over fiber; electrons travel at 10% c over copper; and atoms travel at 0.0001% c over Federal Express. Furthermore, error-correction, compression, and other channel coding techniques also add overhead and jitter to propagation delays. Two major examples are the impact of bandwidth limits and disconnected operation.

Bandwidth. In a related argument from [76], information transmission across space can be viewed equivalent to information storage across time, and thus subject to the same minimum-energy requirements. Since information flux, often termed 'bandwidth,' is also an energy flux, the maximum capacity of channel must be finite. If the desired signal rate is greater than capacity, a buffer could be used to store the input until the output is ready. However, total delay still increases linearly until buffer space is exhausted and, then, data must still be discarded. Furthermore, for store-and-forward networks, buffers on-disk or on-DRAM may be quite large and high bandwidth, but still high-latency. This adds transit time to and from the buffer to the total latency of the channel.

Disconnection. Store-and-forward networks are also robust in the face of limited network partitions. If a line-of-sight is occulted, as for interplanetary networks, or for nomadic cellular users entering a tunnel, the network layer often buffers further to mask the disconnection. System crashes are another source of transient partitions. We choose to model the finite buffer capacity and timeouts for supporting transient disconnection as an additional component of a channel's total maximum latency. Email, for example, typically will be delivered in minutes, but it will not bounce back as undelivered for five days. That makes SMTP a five-day latency channel for our analysis, regardless of what the average latency might be.

2.1.2 Agency

While latency is a physical limit, the concept of an agency is a socially constructed one. We are referring to the divergent interests of the organizations that ultimately own and operate the computers that software runs on, who have different (possibly conflicting) goals and interests [200].

An agency boundary denotes the set of components operating on behalf of a common (human) authority, with the power to establish agreement within that set and disagreement outside it. The anthropocentrism of the concept may seem inappropriate for the concerns of software architecture, at first. The missing link is the tacit assumption that, at least in a capitalist economy, every computing device is owned by a person or organization, and is thus expected to operate on behalf of that agency.

For example, our department recently installed a virus scanner on its central mail server that promptly deleted several drafts of a paper. Those messages' `INFECTED` flags were set solely by the scanner, which is configured, in turn, solely by the support staff. No feature of that centralized scanner software was prepared to brook dissent from its judgment (and furthermore, the attachments were deleted forthwith, in accordance with an entirely separate privacy policy that forbids support staff from storing mail!). In lieu of a decentralized virus scanner that would have let each agent — sender, receiver, administrator — test `INFECTED` locally and react independently, we defeated the scanner entirely by controlling a variable that *was* under our control. It no longer scanned the attachment once we changed its file-type code, the truth of which the scanner entrusts to the sending agency as blithely as it arrogates exclusive control of the `INFECTED` flag for itself.

The generalization of this phenomenon is a 'web of trust.' If every architectural element — every component, every connector, every computer — is controlled by one agency or another, then every connection between them in an architecture diagram must be justified by a corresponding trust relationship between its owners.

Consider a database package. Run a 'local' copy for yourself, and then if you store `x=5`, then `5` is the one and only true value of `x`. Accessing some remote agent's instantiation of the very same software package, though, raises corrosive new concerns about bias and reliability. There is a profound difference between the output of the local database component ("I believe `x` is `5`") and the remote database *service* ("Someone else believed `x` was once `5`").

2.2 Formal Model

Lynch, in [144], claimed "The impossibility of consensus is considered to be one of the most fundamental results of the theory of distributed computing." In 1985, Fischer, Lynch, and Paterson proved that on a completely asynchronous network

(one with maximum message latency $d=\infty$), if even one process can fail, then it is impossible for the remaining processes to come to agreement [71]. Furthermore, Lynch's textbook also includes a proof that even with a partially synchronous network (one with only a finite *d),* but with message loss or reordering, consensus still requires at least d seconds. In general, tolerating f processes failing requires at least $(f+1)$ additional rounds [144].

However, this model of consensus does not ensure simultaneity. Some processes may decide sooner than others, depending on the actual message latencies encountered. Furthermore, if the value being shared is modified, the algorithm ought to be run all over again. In that case, some processes may continue to use the prior value after others have moved on. The following subsection will extend the well-known model of consensus to handle the additional requirements of simultaneity.

2.2.1 Simultaneous Agreement

The term "simultaneous agreement" originated in contract law, specifically for defining financial instruments. An ordinary cash transaction is a simple example of simultaneous agreement upon a trading price, P_{trade}. Computer scientists would call this an atomic transaction, since money is exchanged for goods at the same instant.[4] Another permutation of the same problem is ensuring that every reference to the same variable yields the same value throughout the entire system. Specifically, if a leader's variable X is assigned some value, any follower's reference to it using a local variable Υ must return the same value, or block while waiting to read the current value:

> **Simultaneous agreement** holds between two variables X and Υ whenever $\Upsilon = (X \vee \varnothing)$.

The difficulty arises once X is assigned a new value: How can we ensure that Υ is eventually assigned the same value — and reset to $\varnothing$ before X changes again? Since it takes at least d to establish consensus, simultaneous agreement requires holding the leader's value constant for longer than d. Similarly, tolerating f process stopping failures requires holding the leader's value constant for longer than $(f+1) \cdot d$. A corollary is that it is impossible to guarantee simultaneous agreement for any centralized resource that changes more frequently than $1/d$ times per second (or $1/(f+1)\cdot d$ times per second, in order to tolerate f failures).

To refine our definition of simultaneous agreement more precisely, we need to qualify the value of a variable as a time-variate function by stipulating the existence

[4] Or, at least within the ~100msec threshold of human perception, so that one party can't steal the goods *and* the cash.

of a global clock.[5] We can proceed to define simultaneous agreement as the interval of time satisfying the following conjunction:

$$\exists\, t_o, t_i, t_j : (t_i \le t_j) \wedge (t_i \le t_o + d) :$$
$$\forall\, v : t_i \le v \le t_j : \; P_{leader}(v) = P_{follower}(v)$$
$$\wedge\; \forall\, u : t_o \le u \le t_j : \; P_{leader}(u) = P_{leader}(t_o)$$

That is, the leader and follower values must become equal at some point; and the leader must not have changed in the meantime.

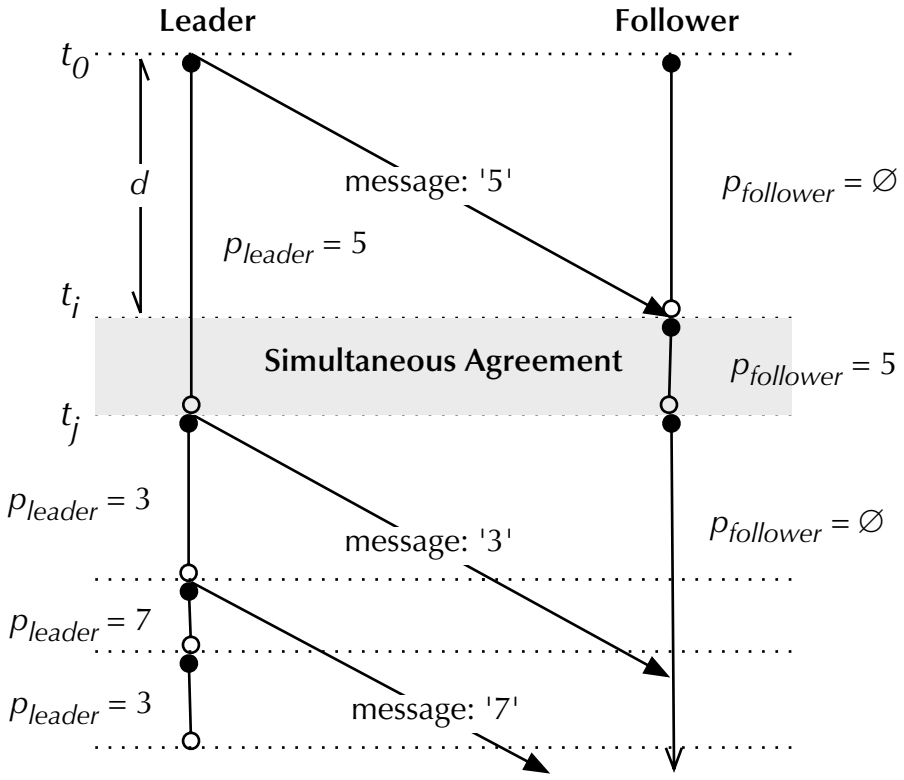

Figure 2: The shaded region illustrates an interval of simultaneous agreement.

This condition is met in the shaded area of Figure 2, where world-lines are drawn vertically for the leader and follower processes; the horizontal separation reflects the time it takes a signal to traverse the distance between them. A message takes at most d to travel, so simultaneous agreement only holds for the span of time after the message arrives until P_{leader} changes from 5 to 3. However, if the variable changes after an interval shorter than d, as from 3 to 7, it is impossible for the message to arrive "in time"; and hence, simultaneous agreement is also impossible.

[5] To be sure, it is only possible to synchronize clocks in an inertial frame of reference. The argument that follows is limited to computers that are at rest with respect to each other.

Note that our definition requires P_{leader} to be constant while agreement is established. At the bottom of the diagram, even though the leader switches back from 7 to 3 before the message "3" arrives at the follower, that is mere coincidence, not simultaneous agreement. Either a follower must contact the leader and request a lock to hold the value constant while it works, or the leader must indicate the period of time it commits to holding a value constant, which is called a lease.

A leased value is represented by a *(value, interval)* pair, where *interval* is specified by the start time and duration of the lease. In conjunction with a global clock, the lease makes it possible for the recipient to determine whether the value is still valid. In Figure 2, the follower can correctly reset the value of $P_{follower}$ to $\varnothing$ at the first instant the leader is able to change (in this case, from 5 to 3) if we extended the contents of the message to specify a lease.

2.2.2 Related Models

Since we are addressing a fundamental aspect of coordinating multiple computers, it is not surprising that there already is a wide variety of related models in the literature. While by no means an exhaustive list, we would like to describe a few and explain how our work is similar to, and different from each.

Byzantine Generals. The Byzantine Generals problem [139] is a variation of the Prisoner's Dilemma [84, 226] where several parties must coordinate an attack simultaneously to win — but they can lie to each other, too. As it was originally described:

> Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement.

The impossibility results derived in that paper have since been superseded by more general results, but the main reason it is relevant to us is that it explicitly poses an equivalence between process failure and what we term agency conflict. By drawing an analogy between an anthropomorphic "traitor" and a mechanical failure, it reveals that the rhetoric of intent can be set aside for a formal abstraction of the problem. The presumed loyalty of "lieutenants" is another assumption that we represent using a hierarchical, bidirectional web-of-trust in our model.

The solution originally proposed is only robust against conspiracies of up to $\frac{1}{3}$ of the generals. This is often sufficient to establish consensus in read-only decentralized systems, if not simultaneous agreement:

> Many P2P storage systems are advertised as repositories of read-only information. The biggest barrier to providing a writable system is consistency—establishing the identity of the latest copy of data, or conversely, that a particular copy is out of date. Such consistency management usually requires a centralized resource to serialize updates. Byzantine Agreement is an ideal distributed serialization technology. [133]

Invariant Boundaries. First identified in [36] as the boundary between systems that agree on an invariant and those that cannot, it closely resembles the specific condition of simultaneous agreement. A more general condition applying to invariant boundaries was stated as the Consistency, Availability, Partitionability (CAP) theorem in [35, 74], and later proven in [96], which states:

> ... it is impossible to reliably provide atomic, consistent data when there are partitions in the network. It is feasible, however, to achieve any two of the three properties: consistency, availability, and partition tolerance. In an asynchronous model, when no clocks are available, the impossibility result is fairly strong: it is impossible to provide consistent data, even allowing stale data to be returned when messages are lost. However, in partially synchronous models it is possible to achieve a practical compromise between consistency and availability. In particular, most real-world systems today are forced to settle with returning "most of the data, most of the time."

This is strongly related to our own approach, though we have chosen to focus on the performance limits of achieving consistency. We will introduce a similar boundary that indicates which components can refer to the value of a variable 'right now,' a boundary which we term the 'now horizon' of an event source in §7.1. In particular, we have linked our definition of the 'now horizon' to a *frequency-domain* model of resources. That is, we can speak of a given horizon restricting the maximum event notification rate of a source; or, conversely, qualify which other software components can reliably refer to an event source with a given minimum update latency.

Logical Clocks. Over the years, there have been many approaches that simulate the operation of a centralized, sequential processor atop a distributed processing network: logical clocks [134], virtual synchrony [26], and group communication [187]. Lamport's original work took some pains to establish the physical basis of the 'Clock Condition':

> We now introduce clocks into the system. We begin with an abstract point of view in which a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred... We now consider what it means for such a system of clocks to be correct. We cannot base our definition of correctness on physical time, since that would require introducing clocks which keep physical time. Our definition must be based on the order in which events occur. The strongest reasonable condition is

> that if an event *a* occurs before another event *b*, then *a* should happen at an earlier time than *b*. [134]

Later, in another work formalizing the mutual exclusion problem, he emphasized:

> ... we have continually used physical reality as our guidepost. (Perhaps this is why hardware designers seem to understand our ideas more easily than computer scientists.) We therefore give a very careful physical justification for all the definitions and axioms in our formalism.
>
> ... formal models of concurrent processes that we know of are based upon the concept of an indivisible atomic operation. The concurrent execution of any two atomic operations is assumed to have the same effect as executing them in some order. However, if two operations can affect one another-for example, if they perform inter-process communication-then implementing them to be atomic is equivalent to making the two operations mutually exclusive. Hence, assuming atomic operations is tantamount to assuming a lower-level solution to the mutual exclusion problem. Any algorithm based upon atomic operations cannot be considered a fundamental solution to the mutual exclusion problem.
>
> ...The reader may find the introduction of special relativity a bit farfetched, since one is rarely, if ever, concerned with systems of processes moving at relativistic velocities relative to one another. However, the relativistic view of time is relevant whenever signal propagation time is not negligibly small compared with the execution time of individual operations, and this is certainly the case in most multiprocess systems. [135]

Like his work, our intent is to remind software architects that there are fundamental physical limits upon today's favored architectural styles: at bottom, they depend on consensus, and consensus is ultimately impossible to assure. That said, we tend to argue *against* models that assign logical round numbers to actions. A key change from the original setting of these works to Internet-scale is that, twenty years later, it *is* possible to have sufficiently accurate physical clocks. Practical clock synchronization using the Global Positioning System (GPS) can achieve time resolutions far more precise than wide-area network messaging latencies [3]. In the "real world," wall clock time is often used to arbitrate access to resources: bidding deadlines, first-in-first-out shipping queues, and so on. All of the architectural styles we will develop rely on a `GlobalClock` component to synchronize lease expirations.

Single-Assignment Variables. An architect could avoid disagreement entirely by restating the problem to eliminate mutable variables [217]. One example is the technique of single-assignment: rather than resetting the value of P_{leader} several times, a series of distinct variables could each be set just once: *First-*P_{leader}, *Second-*P_{leader}, and so on. On the other hand, this model replaces a time-varying function with merely a

series of unconnected observations, it would also rule out a simple user interface that displays "the price is currently *P*."

A common pattern for representing time-varying data with single-assignment variables (PCN, [46, 73]), so-called "futures" (Multi-Lisp, [104]), closures (DREME, [83]), and Hoare's Communicating Sequential Process model (CSP, [107]) is the recursive list data structure. Processes coordinate by waiting for a variable to become defined, but each new value is a pair containing a new value and a new, undefined variable containing the future remainder of the list.

Peer-to-Peer Communication. [95] presents a formal model for peer-to-peer computing that uses variables to represent channels between peers. Casting communication channels as variables may make it clearer that the latency of a network link also determines the maximum possible update frequency of *any* interaction across them. It demonstrates that the limit is independent of bandwidth or pipelining; rather, channel capacity only affect the size of the contents of the variable (its representation).

2.3 Definitions

Our ultimate goal in this dissertation is to understand the implications of decentralization. It thus behooves us to begin with its definition, and definitions of several other terms we will contrast it with later on:

> *Decentralization* (n): the spread of power away from the center to local branches or governments [151].

Perhaps the subtlest aspect of this definition is suggested by the verb 'spread': *one can only decentralize what was once centralized*. Critically, this introduces the stance of an external observer: deciding whether a given phenomenon is centralized, distributed, estimated, or decentralized becomes a matter of perspective. Our goal in this subsection is to provide testable definitions for each type, depending on how control is shared and the degree of certainty required, as elaborated in Table 1.

	Consensus-based	Consensus-free
Master/Slave	Centralized	Estimated
Peer-to-Peer	Distributed	Decentralized

Table 1: Classifying types of variables by control and consensus requirements.

Another way of introducing the terms we will define is by the degree of indirection required. The basic element of information storage is the *value*: centralization requires every agent to use the same value. This can be accomplished as simply as by

connecting several devices to the same wire or other shared medium.[6] The next level of indirection is a private namespace for storing values over time: the *variable.* In a distributed system, a closed group of agents uses a single name to refer to a shared variable — even though its actual value is not stored in one place, but rather in a set of 'shadow' variables held by each participant. Later, in an estimated system, there will still only be one putative shared variable, but the local shadow copy may be a less precise.

These two levels of indirection are fairly well known. In REST, they correspond to the concepts of representation and resource, respectively. The additional distinction our definition of decentralization relies on, though, is a public namespace that may differ across agencies: a *concept*. An intuitive illustration is the difference between typing the URL `HTTP://WWW.WEATHER.ORG/LAX` into a Web browser's address bar, and typing the concept `"LA WEATHER REPORT"` into a Web search engine's query box. The latter may return links to many different organizations' opinions of the weather in Los Angeles — and several more about the weather in Louisiana (which is also abbreviated "LA").

The definitions we will introduce in the remainder of this subsection are all the more important because a single system can exhibit all of these traits simultaneously, depending on the level of abstraction. Consider an online auction service. To a buyer, it appears to be a decentralized marketplace, since anyone can list goods for sale using commonly-understood product descriptions and categories. It also appears to be an estimated service, since the prices displayed in a buyer's browser may be out-of-date or fraudulent. To a seller, it appears to be a centralized auctioneer, since the service alone determines the order and validity of bids. Moreover, to the service's own engineers, it appears to be a distributed database service, since the data is stored and processed on many servers working in parallel.

2.3.1 Centralized

> *Centralized* (adj): drawn toward a center or brought under the control of a central authority [151].

Applied within the context of software architecture, centralization can be described as the practice of assigning exclusive control to modify information or take action. Two examples of centralized software architecture are the use of a shared database in the client/server style, or the use of a single mouse input in an event-driven user interface style. Many other components will rely on references to the resulting database record or cursor position, respectively. Nevertheless, those resources are the only authoritative sources of such information.

[6] "A wire is just a renaming device for variables."
— Alain Martin, asynchronous VLSI designer [146]

> A variable X is considered **centralized** if and only if X is modifiable at only one location, and all references to X from other locations require simultaneous agreement.

By location, we mean something more specific than the street address of a computer. Ultimately, information is represented by specific arbiters: physical devices than can make an exclusive choice between representations of information, such as a hole on a punched-card, a magnetic particle on disk, or a flip-flop circuit on a chip.

If we permit information in a follower arbiter to be used in a computation without establishing simultaneous agreement with the leader arbiter, we contradict the definition of centralization by permitting the same computation at the same time, depending on the same centralized variable, to yield different results at different locations.

The following additional rule is an elementary result of queuing theory, but bears restating: if the request rate exceeds the service rate, total service time increases linearly. Therefore, given that d is the maximum latency from the leader to any follower:

> A centralized variable requires waiting at least d seconds between updates.

Note that this limit is entirely independent of computing speed or bandwidth. Consequently, no centralized software architecture can process events occurring more frequently than $1/d$ times per second (where d is the maximum latency between components). This limit can cause serious problems if a software architect intends that a centralized variable should represent some phenomenon in the "real world" that occurs more rapidly.

2.3.2 Distributed

> *Distributed* (adj): spread out or scattered about or divided up [151].

Sharing control between several agencies requires the concept of a variable that can be modified at several locations, not just one. However, since arbiters at each location are physically separate devices, distribution must be defined as a condition over several "shadow" variables rather than a new kind of variable in and of itself.

To be sure, many popular models for distribution merely extend the single leader/multiple follower model of centralization by delegating control to another leader temporarily (see §2.3.2.1 below). Even two-phase commit protocols for 'distributed' databases rely on a centralized transaction monitor. Similarly, collaborative document repositories that use locking delegate control over a version to a client for

a limited time. Not surprisingly, this ersatz form of distribution — merely letting clients propose new values for a variable still controlled by a single server — offers the same performance as any other centralized system.

A genuinely non-hierarchical (peer-to-peer) approach to distributing control over the value of a shared variable, instead, is to apply a shared decision function to an ensemble of variables owned by each participating agency. We believe that is a more appropriate model of how power is distributed in society, or in software. Each individual agent only controls a private preference, but the distributed consensus value is established by some shared decision function, such as the mean, mode, maximum, minimum, or majority.

It follows that since the function *df*()is identical throughout, and its inputs are local variables that are each in simultaneous agreement with their respective remote leaders, then every local result of applying *df*() will also be in simultaneous agreement:

> A variable Υ is considered **distributed** if all references to Υ are replaced by the result of applying a shared decision function *df*() over a series of simultaneously-agreed variables X_1, X_2, ...

The minimum time it takes to calculate Υ is at least the maximum latency between *any* two locations in the system. Since simultaneous agreement with a centralized variable requires waiting for the maximum delay from the leader to any other follower, it follows that simultaneous agreement on an ensemble of centralized variables requires at least the maximum delay from each possible leader to every possible follower in the system.

However, actually realizing the lower-bound time of $(f + 1) \cdot d$ is more difficult than in the centralized case, since it requires strict clock synchronization and phase alignment on every update. Only if every centralized variable changes at the same instant, and for the same lease duration, can we guarantee that all agents have received all updates after only d seconds have elapsed. Note that if some of the input variables, in turn, are also distributed, it need not increase the minimum time bound; all of the decision functions in the entire system could still evaluated as soon as all the inputs are in simultaneous agreement.

The upper bound is the more relevant, and more involved, calculation. Without loss of generality, we choose to ignore fault-tolerance (that factor of f above). Next, we choose to model the modification of a centralized variable as an event with a lease. Assigning a new value to variable v at time t implies that the value cannot be modified during the interval $[t, t + L_{min})$, where L is the lifetime of the lease. Since it takes d to propagate information, it follows that simultaneous agreement is definitely

feasible during the interval $[t + d, t + L_{min})$. That interval is obviously nonempty, since we already concluded that L_{min} must be greater than d in §2.2.1.

Simultaneous agreement over the distributed variable Y is only possible during the intersection of all the intervals where simultaneous agreement also holds for each variable $X_1, X_2, ..., X_N$. Without loss of generality, we choose to sort the time at which each of those variables are assigned values in ascending order as $t_1, t_2, ..., t_N$. Thus, simultaneous agreement over Y holds only during the intersection:

$$\forall\ i : 1 \le i \le N:\ \cap\ [t + d, t_i + L_{min})$$

Since, in general, the intersection of several intervals of the form `[MINIMUM, MAXIMUM)` is the interval `[MAX(MINIMA), MIN(MAXIMA))`, that is equal to:

$$[t_N + d\ ,\ t_1 + L_{min})$$

For that range to be nonempty,

$$L_{min} > (t_N - t_1) + d$$

Since the times are a sorted sequence, the value of $(t_N - t_1)$ can be expanded into the sum of the differences between each successive time:

$$(t_N - t_{N-1}) + ... + (t_3 - t_2) + (t_2 - t_1)$$

This is at most $(N - 1) \cdot \max(t_k - t_{k-1})$. The crux of the following calculation is that it is not possible for successive times to be more than d apart.[7] Informally, we would already be in simultaneous agreement with a variable that had not changed for at least d; we should have included the *prior* modification of that variable in our sort order.

Formally, if $t_{k+1} - t_k \ge d$, then the prior modification time of variable X_{k+1}, called t_{k+1}', is at least $t_{k+1} - L_{min}$ and at most $t_{k+1} - L_{max}$. This implies simultaneous agreement must be possible during the interval $[t_k + d, t_{k+1})$, contradicting the sort order by including t_{k+1}' instead. Thus,

$$L_{min} > (N - 1) \cdot d + d$$

Therefore, we conclude that a variable under the distributed control of N agents takes at most $N \cdot d$ to modify. We can also state this as a limit on the update frequency of a distributed variable:

---

[7] More precisely, though, they *could* be more than d apart if some of the inputs are also, in turn, distributed variables. While recursive composition of distributed variables does not increase the lower bound, it does increase the upper bound. $\max(t_k - t_{k-1})$ is actually the minimum lease between updates to variable X_k, which we have shown is d for centralized variables and will show is $N \cdot d$ for distributed variables.

For example, if the two-way distributed variable F depended on one centralized variable, G, and one three-way distributed variable, H, then the total minimum delay between changes to F would be $4d$, rather than $2d$ (where d has its usual definition as maximum network latency).

> An N-way distributed variable requires waiting at least $N \cdot d$ seconds between updates.

Note that this limit is also entirely independent of computing speed or bandwidth; it affects all consensus-based architectural styles. Consequently, no distributed software architecture can process events occurring more frequently than $1/N \cdot d$ times per second.

2.3.2.1 Delegated

It *is* possible to choose decision functions that allow "distributed" variables to updated far more rapidly. Rather than outputting the actual consensus value, the function decides *which* agent is (temporarily) in charge. Such delegation of authority to a single central location requires distributed mutual exclusion protocols. Such solutions can share control more rapidly, but require agents to wait (block) before updating local variables.

This is why our $N \cdot d$ finding above differs so significantly from today's popular understanding of "distribution." If we choose a monotonic decision function, such as *oldest()*; and require the input values to also change monotonically, say by time-stamping each update request; then the dependency-chaining argument used above to derive that factor of N falls out.

Consider the effect of choosing a decision function that implements a distributed-mutual-exclusion protocol such as Petersen's algorithm [180] or Lamport's Bakery algorithm [136]. In such turn-taking solutions, there is no way to cancel attempts to acquire a lock. This means the input values of these decision functions cannot change arbitrarily. Specifically, later changes to an input variable cannot possibly change the output decision. A simple example is requiring monotonically-increasing time stamps in order to choose the "earliest" lock request.

In such cases, while the maximum delay for *some* client to change the shared value may be as little as $2d$, the maximum delay for a *particular* client to change the value of the shared resource may be unbounded. Even when using a 'fair' mutual exclusion algorithm, it can require waiting for all N others to acquire and release their locks first.

Therefore, we hasten to add this caveat to our result above: it *is* possible for a distributed system to process events occurring more frequently than $1/N \cdot d$ times per second, indeed, as fast as $1/2 \cdot d$ times per second; but only *if* changes to local variables can be arbitrarily delayed, by as long as $N \cdot d$ seconds.

2.3.3 Estimated

Estimated (adj): calculated approximately [151].

Once simultaneous agreement becomes impossible to guarantee 100% of the time, the next step is relaxing the probability to some lower percentage before finally resorting to a completely decentralized solution at 0%. We refer to this broad middle range between consensus-based and consensus-free systems as an estimated system, since it relies on trading off precision for update frequency in favor of establishing at least approximate agreement.

The crux of such substitution is a value-laden judgment about the threshold probability that constitutes "approximate agreement."[8] The reason we add this complication is that each agency has its own purposes when it refers to a shared concept. For example, one may need to know the price to the nearest penny, while another might be satisfied by ±50%. For another example, a buyer's estimate may need to err on the high side of prices, while a seller's application needs low-ball estimates. Both of these are forms of confidence-interval estimates, as opposed to point-value estimates.

Furthermore, the actual predictability of current values from past data is limited, especially as latency increases. Some values are simply more auto-correlated than others: consider the difference between measurements of "outside temperature" and "subject line of the latest email."

Nonetheless, we intend to discuss estimated systems in contrast with decentralized systems because they correspond to a distinct architectural choice. Estimates extend the master-slave pattern of control by presuming that a 'true value' exists to establish simultaneous agreement with: even though our measurement of the bookstore's inventory may be imprecise, it is still not under the customer's control. In the next subsection, we will contrast this with the peer-to-peer pattern where variables will be replaced by decision functions — but in that case, private assessment functions rather than the shared ones used in distributed systems.

We begin our formal definition of an estimated variable with a declarative relationship between two variables:

> A variable E is considered to **approximately agree** with another variable X with probability $P\%$ if, over a sufficiently long period of time, the ratio of the time $E(t) = X(t)$ out of the time $E(t) \neq \varnothing$ is $\geq P$.

In other words, $P\%$ of the time that the estimate is defined, it must be in simultaneous agreement with the leader. The next step is to acknowledge that it takes

[8] Technically, 'approximate agreement' applies specifically to consensus over real-numbered values when using multiple-round variations of the algorithm to converge on a decision [144].

time to transmit information by positing a deterministic estimator function to recover present values from past data:

> A function *ef()* is considered an **estimator** of another variable *X* with probability *P%* if its only input is past observations of the value of *X*, and its output approximately agrees with *X* with probability *P%*.

Before we can use this to provide an operational definition of an estimated variable that can actually be implemented, though, we need to depart from our model of consensus-based variables by adding the notion of ownership. That is, the value of any consensus-free variable is both time-variate *and agency-variate* — it not only depends not only upon when you ask, but also upon whom you ask:

> An **owned variable** *V* is an *(V(time), Agency)* pair, or alternatively, $V_A(t)$.
> An **owned function** *f()* is an *(f(), Agency)* pair, or alternatively, $f_A()$. Its result is owned by *A*.
>
> A **trust relationship** between two agencies *A* and *B* holds whenever *A*'s value for some variable *V* is identical to *B*'s. In that case, we say "*A* trusts *B*"; if *A* can also modify the value of *N*, then "*B* trusts *A*" as well.
> A **web of trust** *T* for a variable *V* is a directed graph over the set of all agencies, where each edge represents a trust relationship.
>
> A **trusted invocation** of an owned function $f_A()$ is one that is applied to several inputs $V_B, V_C, V_D, \ldots$ if and only if there exists a path from *A to* each of *B, C, D, ...* in *T*.

Taken together, these definitions expand the concept of location used earlier into an agency boundary. 'Local' and 'remote' now refer to whether a variable is owned by the same agency that owns the function processing it, rather than a relationship in space. Now we can specify how to compute a local estimate of a trusted remote resource:

> An owned variable X_A is considered **estimated** by agency *B* with probability *P%* if every reference to $X_A(t)$ by *B* is replaced by the trusted invocation of $ef_B(X_A(t-\lambda))$, where $ef_B()$ is *B*'s estimator, there is a path from *B* to *A* in *T*, and λ is a positive value representing transmission latency.

Note that this definition does not necessarily produce estimates that the original owner *(A)* would trust.[9] It is sufficient that the estimate produced is substitutable for *B*'s purposes.

Admittedly, these definitions do not provide guidance on how to actually implement an estimator function. Almost any interesting estimator would be very domain-specific, such as a dead-reckoning position predictor for a given kind of vehicle, or a historically-informed model of interest rates. It also does not speak to the complication of specifying 'confidence intervals' for discrete variables: what would be the probable images for an occasionally-connected security camera? There are also estimators that summarize multiple updates: typing faster than a GUI application is prepared to respond can create synthetic events that coalesce those keystrokes into a single string.

Nevertheless, the most commonly encountered estimator is the null estimator: stasis. An ordinary Web browser that continues to display a cached page even after the server operator changes it is one such example. The probability of approximate agreement using the null estimator is readily calculable: delay since the last update received divided by the average interval between updates. Polling a news site every five minutes that changes hourly, on average, yields results that are in simultaneous agreement (1 - $\frac{5}{60}$) = 91.67% of the time.

2.3.4 Decentralized

> *Decentralized* (adj): withdrawn from a center or place of concentration; especially having power or function dispersed from a central to local authorities [151].

Decentralization is fundamentally different from either centralization or distribution because it permits multiple, distinct, simultaneously valid outcomes for the same decision. It also differs significantly from estimation, since several independent agents with their own estimates of a centralized or distributed variable can end up using multiple, distinct values at the same time, they are all attempting to recover a single outcome for the shared decision. In a consensus-free system, each agency's updates merely represent its own opinion — and comparing more opinions from other, trusted agencies can increase the accuracy of one's own opinion. The statistical premise behind this is that the errors of each agency are independent: they can't

[9] Of course, *T* can also be dynamic. Trust relationships can be added or deleted at any time — and that since it takes time to propagate trust changes, untrusted data can still sneak in. Such dynamism could be called out by referring to *T(t)* throughout, as well as subscripting the web-of-trust for each variable, too. In the interest of simplicity, we do not add such distinctions in our notation.

all conspire against you. Thus, a hallmark of decentralization is its specific emphasis on agency conflicts — a shift from matters of *fact* to matters of *opinion*.

Of course, if there is no causal connection between each agency's own decisions, a decentralized system degenerates into a set of isolated centralized or distributed systems. Consider how independent copies of a spreadsheet programming running on separate PCs have their own values for cell `A1`: without positing any cross-references between those spreadsheets, the distinction that `A1` happens to have multiple, distinct simultaneously valid outcomes on each PC is moot.

The interesting case for decentralization is when there is partial coordination between peers about the "same" decision. To determine sameness, decentralized concepts must be defined by each agency with respect to a shared namespace. This stands in contrast to the shared decision function and mutually-trusted set of peers in the case of distributed variables.

Moving from informal discussion towards a formal definition, we begin by noting that a decentralized variable is not owned by any single agency (otherwise, it would have a 'true' value). Instead, it is identified by a name that is owned by some agency that all other agencies trust, at least to fix the syntax by which individual agencies make representations about the "same" decision. We thus posit that there is a bottom ($\perp$) of T, such that a path exists from every vertex in T to $\perp$.

> A **namespace** is a variable owned by $\perp$ that contains a mapping from each name to a list of the owned variables representing that name. Alternatively, $namespace_{\perp}(name, t) = \{variables\}$.
>
> A set of **correspondent variables** for a given agency A and a *name* is the set of variables representing *name* that A trusts:
>
> $$correspondents_A(name, t) = \{X_B : X_B \in namespace_{\perp}(name, t) : \exists \text{ path } A \rightarrow B \text{ in } T\}$$

The next step is to form an assessment of the opinions held by the panel of correspondent agencies, based on the best available estimate or simultaneously agreed representation of agency's opinion. However, since a decentralized variable's value is not even required to be in approximate agreement with any other variable, we cannot specify a declarative condition that defines its validity. Therefore, we can only offer this operational definition, based on an agency-specific decision function that can take into account the value of multiple remote resources:

A function *af()* is considered an **assessor** of a set of correspondent variables if its input includes past observations of any variables from that set.

An owned variable $X_\perp$ is considered **decentralized** according to agency A if every reference to $X_\perp$ by A is replaced by the trusted invocation of $af_A(ef_A() \bullet \{correspondents_A(\text{“X”}, (t-\lambda)\})$, where $af_A()$ is A's assessor, and $ef_A()$ is A's estimator for each of the variables in the set of correspondent variables, and λ is a positive value representing transmission latency.

Or, in less cryptic terms, a decentralized concept is represented by a hypothetical variable owned by “nobody.” To actually refer to that variable in practice, an agency must first determine which other agencies' opinions it trusts; second, try to determine what those opinions actually are, by using an estimation function to recover its likely current value from past data; and finally, assess the entire panel of other opinions according to its own policy.

2.3.5 Resource

So far, we have been describing the properties of abstract, mathematical variables. Our bridge between theory and practice is be an architectural element from REST known as a ‘resource.’

Consider how an architect would approach the challenge of actually using a centralized, distributed, estimated, or decentralized variable to store, say, the ¥/$ exchange rate in a multi-agent trading application.

The first challenge is for each component in the application to refer to the same resource consistently. This is a naming problem: an additional layer of indirection that uses a common symbol to refer to the value we are seeking consensus over. Following our definition of an owned variable, a useful name syntax should couple an agency identifier with every variable identifier — similar to how a Uniform Resource Identifier (URI, [22]) concatenates a domain name and path name. The essential reason for this level of indirection is time-independence. Since we are dealing with time-variate variables, we consider each state change an ephemeral event notification about an event source with a persistent name.

Our requirements for manipulating a named variable correspond to the definition of a resource in REST:

> The key abstraction of information in REST is a *resource*. Any information that can be named can be a resource: a document or image, a temporal service (e.g. “today's weather in Los Angeles”), a collection of other resources, a non-virtual object (e.g. a person), and so on. ... A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

> More precisely, a resource R is a temporally varying membership function $M_R(t)$, which for time t maps to a set of entities, or values, which are equivalent. The values in the set may be *resource representations* and/or *resource identifiers*.... Some resources are static in the sense that, when examined at any time after their creation, they always correspond to the same value set. Others have a high degree of variance in their value over time. The only thing that is required to be static for a resource is the semantics of the mapping, since the semantics is what distinguishes one resource from another. [67]

We believe that our hypothetical architect needs centralized, distributed, estimated, or decentralized *resources*, not variables alone. Adapting our formal model of variables to resources requires recognizing that the object of agreement can also be a function, not just a value.

In general, the equivalence of functions and values seems obvious: on any von Neumann architecture computer, where programs are also stored as data, so we only need to refine our model of simultaneous agreement slightly. Specifically, we choose to model a named variable as a time-varying membership function that permits multiple, simultaneously valid representation formats for the same variable:

Simple agreement over a resource named R between a *leader* and a *follower* separated by a positive transmission latency λ is defined as:

$$\forall\ t : t \geq 0 : M_{R@follower}(t + \lambda) \subseteq M_{R@leader}(t)$$

This definition still permits the follower's membership function to fail. If, say, the network connection to a remote leader were interrupted, the resource could yield $\varnothing$, the empty set of representations. The additional constraint for simultaneity would follow the line of argument in §2.2.1.

2.3.6 Representation

The second challenge our architect faces is format-independence. In a large-scale software development project, it is unreasonable to expect that every component will use identical data formats, operating systems, programming languages, and so on. A resource can be made concrete using many equivalent formats of the same abstract state: floating-point, text, pictures, audio, and so on. Thus, the second architectural element from REST that we need is a 'representation':

> REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A *representation* is a sequence of bytes, plus *representation metadata* to describe those bytes. [67]

Specifically, there are two key pieces of metadata from the definition of REST we choose elevate for consideration in our own abstract model of representations: the resource identifier and cache lifetime. These are encoded in four headers in

HTTP/1.1: HOST, CONTENT-LOCATION, DATE, and EXPIRES. We relabel these for our purposes as the agency, name, and lease of a value (content):

> A **representation** is a 4-tuple *(A, N, V, L)*, consisting of *A,* the agency making the representation; *N,* the name of the variable being represented; *V,* its claimed value; and *L,* a lease indicating the interval of time when *A* can replace references to *N* with value *V*.

This model also permits us to define four types of representations that correspond to our four types of variables: centralized, distributed, estimated, and decentralized. Table 2 summarizes the rules that determine whether a local proxy variable can be assigned the value found in a given representation tuple: based upon whether it was made by self or others; whether it is currently valid or expired; and whether there can be more than one simultaneously valid representation for the same name.

	Valid Lease	Expired Lease	
Same Agency	$(self, R, v, now)$	$ef_{self}(self, R, v, past)$	Master/Slave
Other Agency	$df_{shared}(others, R, v_n, now)$	$af_{self}(others, R, v_n, past)$	Peer-to-Peer
	Consensus-based	Consensus-free	

Table 2: Rules for selecting valid representations for each type of resource.

Table 2 also lets us speak of *facts* and *opinions*. Facts are representations made by oneself, or by another trusted agency; opinions are a label for representations made by any other trusted agency (untrusted strangers would be ignored entirely). It also permits us to describe the three kinds of functions used earlier in our definitions of distributed, estimated, and decentralized variables in new ways. A shared decision function is used to consolidate one representation out of many different agencies' facts. A private *ef()* is used to derive current (estimated) representations from old facts. Finally, a private *af()* is used to derive a current representation from many different agencies' opinions, old or new.

2.4 Problem Statement

The preceding analysis enables us to state the problems we intend to address in this dissertation more precisely. We begin by decomposing our investigation of software development for decentralized systems into a series of three sub-problems: definition, design, and implementation.

First problem: What is the nature of "decentralization"? What we have found so far is that to understand decentralization, we also have to define centralization,

distribution, and estimation formally. We also needed to apply those abstract definitions to identify corresponding properties of corresponding architectural constructs. Thus, we can restate our first question as:

> What are the properties of centralized, distributed, estimated, and decentralized variables and resources?

Second problem: What architectural styles can enforce the properties that we just identified for centralized, distributed, estimated, and decentralized systems, respectively? For reasons we will discuss in the next chapter, we chose to begin with the REST architectural style. This allowed us to restate our search for new styles as an effort to extend an existing style:

> What new architectural elements and constraints can be added to REST to derive new architectural styles that support centralized, estimated, distributed, and decentralized resources?

Third problem: Are such styles practical to implement and apply? Note that we are not tackling the more ambitious questions of scalability or performance. Our initial contribution is simply to establish that these new styles and applications are possible. Similarly, while our styles are derivatives of REST, and even our infrastructure implementations are derivatives of corresponding REST infrastructure, such congruence is, in itself, not necessarily a contribution. Developing sample applications will also require us to propose at least an initial design methodology for decentralized applications in due course. Forgoing such elaborations, our third problem can be restated as:

> Are there practical implementations for each new architectural element and constraint? Are these new styles usable for designing centralized, distributed, estimated, and decentralized applications?

We begin our investigation of these three problems by summarizing past research into related architectural approaches to decentralization in the next chapter.

Chapter 3: Architectural Approaches to Decentralization

Our investigation of architectural approaches to decentralization begins by considering related work from three different research communities: software architecture *per se*; middleware technology; and precedents from internetworking.

3.1 Software Architecture

We begin by tracing the emergence of the very concept of software architecture. The systematic study of software system architectures, as a discipline, is relatively new, beginning in the early 1990s, and rapidly evolving [1, 88, 91, 179, 195, 202, 204, 229]. It has described a wide variety of techniques for designing, developing, and maintaining software systems.

Within this literature, researchers have been describing ways to constrain particular application architectures to enforce certain desired properties of the entire system. Such concerns coalesced around the notion of an *architectural style* [85, 89, 149, 160, 179]. An example is the popular Client/Server architectural style, which enforces consensus by construction, because all operations on a shared resource must be processed as transactions against a single, centralized database[10].

Defining these terms more precisely can be a contentious exercise, so we chose to adopt Fielding's definitions in [67], which are based on local usage at UC Irvine:

> A **software architecture** is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.
>
> An **architectural style** is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.

To take a closer look at these ideas, it is necessary to enumerate the sort of "run-time elements" found in a typical software system: components and connectors. Again, per [67]:

> A **component** is an abstract unit of software instructions and internal state that provides a transformation of data via its interface.

[10] Even if it were physically implemented on a distributed cluster of servers, we would consider that a logically-centralized database.

> A **connector** is an abstract mechanism that mediates communication, coordination, or cooperation among components.

Note that these are suitably generic levels of abstraction. A component may be *internally* constructed using an imperative, functional, or object-oriented programming language, but choosing an implementation approach does not extend to describing the architecture of the overall system. Similarly, a connector may be implemented by function calls with arguments on a stack, or by using a network messaging protocol, but neither choice deems the entire system "RPC-style" or "event-driven."

Of course, there are more complex architectural elements that cannot be considered exclusively either as components or connectors. Consider the case of a filter that converts value from one currency to another. Depending on the level of abstraction at issue, the converter can be seen as a component, containing a complex and configurable program; or it could be seen as a mere connector that is translating presentation information for a localized user interface. Technically, the definition of connectors in [67] would classify such a filter as a component:

> Perhaps the best way to think about connectors is to contrast them with components. Connectors enable communication between components by transferring data elements from one interface to another *without changing the data* [emphasis added]. Internally, a connector may consist of a subsystem of components that transform the data for transfer, perform the transfer, and then reverse the transformation for delivery. However, the external behavioral abstraction captured by the architecture ignores those details. In contrast, a component may, but not always will, transform data from the external perspective.

Traditionally, "without changing the data" would be taken to exclude operations such as discarding, summarizing, or even predicting new data. However, at some layers of abstraction, converting a price from dollars to euros does not constitute a "change"; it is merely a different representation of the same price. For another example, a service that computed a running tally of a stream of votes would have to be considered a component, since it "transforms data"; but we might consider it a connector if its ultimate purpose is to ensure that message traffic never exceeds available bandwidth.

In our own work, we favor describing such services as connectors, because on a decentralized, error-prone network, there is no possibility of guaranteeing reliable message delivery — so the very goal of transparent connectivity is questionable. For example, we will note later on that message loss due to network error is indistinguishable from message deletion pursuant to security policies, and hence we will refer to a `TrustManager` "connector" in our analysis.

3.1.1 Example: C2

As an example of an architectural style, consider how the C2 event-based style works [213]. It is a useful example since it is distinct from both popular understanding of the broad term Client/Server, and the REST style we will be working so extensively with later on.

Decentralization has often been equated with "loose coupling," which in turn is often claimed as a virtue of event-driven architectural styles. For example, [42] makes just such a claim, albeit in reverse order:

> The components of a loosely coupled system are designed to generate and respond to asynchronous events

Intuitively, an event-based style ought to be better prepared to react to changes occurring "out there," in the environment. An archetypal example is the inverted control loop typical of Graphical User Interfaces (GUIs). Rather than programming an application to proceed down a fixed path, demanding human input under program control, GUI libraries divide programs into finer-grained 'event-handlers' that are invoked on demand by human input. A classic example of this is the Model-View-Controller architectural style introduced by Reenskaug as part of the Smalltalk project in 1979 [129].

Of course, there are many different architectural styles that constrain how event handlers behave. C2 claims to use event notification to support dynamic evolution of applications by eliminating one form of coupling between components:

> The C2 style supports the development of distributed, dynamic applications by focusing on structured use of connectors to obtain substrate independence. C2 applications rely on asynchronous notification of state changes and request messages. [67]

By defining the only possible communications flowing "down" to lower layers as notifications, a type of message that is purely advisory (no assured consequences), C2 does not permit component designers to rely on any particular lower-layer behavior. Of course, it is still possible, even common, to choose to publish 'notifications' that are actually requests in disguise, but at least the C2 style makes clear the costs of adding hidden dependencies that violate its intended acyclic graph constraint.

C2 also includes some features that support decentralization. Its use of independent components with no shared memory assumption implies that each component must maintain its own state, and a perception of the state of other components, to function and make decisions. Its use of multicast events decouples individual components from one another because event sources and destinations are not named — all message routing is done by the connectors. C2's ability to add and

remove components from a C2 system at runtime, including components not anticipated by the original system designers, is a necessary feature of decentralized multi-agency systems. On the other hand, the entire integration is assumed to be within a common trust domain. Only recently has our group considered enforcing information hiding between different users' components, such as a battle management scenario requiring selective information flow, even between allied forces.

Other investigators have also studied architectures built around events for integrating applications. Rapide [143] is an architectural style and tool-set that focuses on formally analyzing and simulating architectures that communicate via asynchronous events. However, requiring an entire application, developed out of parts built by several independent organizations, to be described in a consistent notation suitable for third-party static analysis may prove unrealistic [142]. Design-time static analysis also does not adequately accomodate the dynamic nature of decentralized software evolution [177].

3.2 Message-Oriented Middleware

The term 'middleware' emerged by a process of exclusion: it can refer to any software product used by application developers that is not already part of an operating system, ranging from databases to performance monitors. It thus came to include the infrastructure for connecting components in both event-based and network-based architectural styles, sometimes known as "integration middleware."

Its earliest origins are in the form of message queues, which decoupled components that executed at different times (batching). The message bus was the next logical phase, decoupling multiple components in "space" — so that all components could participate on an equal basis. Similarly, message brokering added an additional layer of indirection, where message destinations were selected dynamically. The subtle difference of the latest phase, message routing, is its presumption there *already* are multiple, distinct message passing-networks to interconnect.

There are two dominant forms of middleware in use today. One sort traces its roots to Remote Procedure Calls (RPC, [27]), such as CORBA [171, 172, 205] and COM/DCOM [38, 48, 231]. The others are lumped together under the heading of Message-Oriented Middleware (MOM, [119]), such as IBM's WebSphere MQ [13], Tibco's Information Bus [174, 184], and myriad Java Message Service (JMS) [9, 113, 161] implementations. MOM arguably even encompasses research into event notification services [16, 41, 125, 150, 168, 191, 196, 227], instant messaging protocols [56, 153, 165], and tuplespaces [8, 112].

RPC-based middleware is insufficient for decentralized application development. In presumes tight coupling between components, since the request-response paradigm that implies the establishment of consensus, and the use of somewhat

arbitrary, fixed APIs as the only method of accessing another component rule it out as a viable candidate for supporting decentralized systems [27].

Message-oriented middleware, on the other hand, has features that are much closer to those needed by decentralized systems. Typically, messages are one-way, can be delivered to multiple recipients implicitly, and can accommodate generic data types. Many MOM systems support several different patterns of message transfer: publish/subscribe or point-to-point, address-based or content-based, lossy or reliable, sequenced or best-effort, and push or pull delivery.

Of course, there are also approaches which bridge between both categories. For example, Message-Driven JavaBeans [147] enable event-based communication with components that only have an RPC interface. This is an example of transforming an explicit invocation into an implicit one [90].

3.3 Internetworking

Decentralized systems must permit failure: crashed components, partitioned networks, congested channels, and myriad other sources of uncertainty. Fault-tolerance became the key motivation for *implicit signaling,* the key difference between centralized, circuit-switched data networks and the decentralized, packet-switched Internet. Rather than running an extra signaling link alongside each pathway to indicate whether a link is available, broken, or congested — enabling an 'intelligent' network core to control data flows — the Internet chose to rely only on a 'best-effort' network that, seemingly paradoxically, did not have *any* way of signaling why packets were lost. Nonetheless, it is still possible to achieve high levels of reliability even over such a lossy network.

The solution was decentralizing network control from the core to the edges, an approach termed "the end-to-end hypothesis" [49, 186, 198]. In particular, TCP was layered on top of IP[11] using a sliding-window protocol to arbitrate access amongst decentralized nodes fairly [43]. Today, even most "reliable" MOM services actually rely on TCP's end-to-end retransmission to operate across the public Internet. We plan to extend that analogy further to show how many of the different MOM modes can be supported on top of a simple, best-effort, one-way event notification service.

Another even more radical application of the end-to-end hypothesis is to extend the best-effort model to *reduce* latency using speculative transmission. In Mirage [218], for example, since the state the remote receiver will be in when a packet arrives is uncertain, "extra" packets are sent along that react to those likely future states, to take advantage of spare capacity afforded by then-novel gigabit networks.

A secondary observation from the internetworking research literature is that, in many ways, the Internet remains quite centralized. Its correct operation relies on

[11] Actually, IP was only separated "out of" TCP after the fact, in 1978 [208].

hierarchical control of the Domain Name System (DNS) and IP address allocation. Its applications also make egregious trust assumptions, such as the validity of any `FROM:` address in SMTP, the root cause of e-mail spam.

Furthermore, another response to the threat of inconsistency and attack has been to re-centralize some intelligence into the core of a so-called 'active' network [214, 215]. Requiring IP routers to execute arbitrary programs to sort, filter, and prioritize traffic flows seems may seem at odds with the end-to-end principle's defense of 'dumb' networks, but it is still in line with the social reality of Internet access today: most users must simply trust their ISPs. After all, decentralization can only benefit a node with more than one link to the whole network; a leaf node's connectivity is centralized *de facto*, otherwise.

Chapter 4: Understanding REST

Before we proceed to extend REST to accommodate centralized, distributed, estimated, and decentralized resources, it behooves us to understand the properties that REST can induce on its own.

The origin of REST is intimately related to the development of the World Wide Web. It purports to explain the design of the most successful decentralized application to date. Nonetheless, REST is not as suitable for developing decentralized applications as is often assumed; it does not even specify a reliable write mechanism, much less a model for asynchronous event notification. On the other hand, our own proposed styles will be reusing and refining many of REST's existing features to support decentralization, such as caching and security.

The remainder of this chapter will introduce the structure of our arguments for specifying, defining, validating, and implementing new architectural styles. Using REST as an example, we will show why inducing consensus leads us to introduce a new architectural element, the GlobalClock component, in order to strengthen the original style's latent requirement for clock synchronization.

4.1 Modern Web Architecture

The World Wide Web is arguably the most successful decentralized information system of the 20th century [23]. It immediately enabled users to read, write, and navigate hypertext documents at global scale [21]. With additional effort, it enabled 'virtual enterprises' to deploy complex applications and manage collaborative workflows [69]. While there were many "accidental" reasons for its rapid adoption — installation without administrator privileges, ease of migrating existing files and directories, flexible scripting interfaces and more — its essential advantage was accommodating *broken* links [121]. The Web gracefully tolerated inconsistencies arising from users' freedom to create links to others' works without prior coordination.

The technological foundations for the Web's success were laid decades earlier, particularly the (wholly centralized) Domain Name System (DNS, [156]), which made it so easy to identify external agencies. Nevertheless, the Web's success cannot be ascribed to the usual infrastructure for developing client/server applications across the Internet — too many other similar systems had failed already to reach that conclusion (such as WAIS [115], Hyper-G [148], and Gopher [12]).

Thus, while "decentralization" is often equated with merely separating software components across a network, the Web's success as a decentralized system merits explanation beyond the benefits of networking alone. We believe the key was adding

another layer of indirection[12] between abstract resources and concrete representations.

Depending on the form of the request (as specified in HTTP headers), a representation of the requested resource is returned to the client from the server or proxy. This representation may be trivial — the contents of a static file on the server, or more complex — those contents transformed in some way (consider a dynamic web page or one that is translated into a different language based on the client browser's locale). Constraining the architectures of Web applications (and, indeed, the Web itself) in these ways has induced several beneficial properties: the Web is highly scalable, extensible, and interoperable with many different types of devices and data formats.

4.1.1 An Architectural Style for the Web

There are many different network-based architectural styles, such as client-server and remote-data-access [67]. The style popularly known as "3-tier client/server" is a combination of both of those styles: presentation interface at a client, business logic on a server, and storage in a database. It was arguably the dominant style of application construction before the Web became popular in the mid-90s.

REpresentational state transfer (REST) was intended to explain the modern Web architecture [70] that eclipsed it:

> REST is an architecture that separates server implementation from the client's perception of resources, scales well with large numbers of clients, enables transfer of data in streams of unlimited size and type, supports intermediaries (proxies and gateways) as data transformation and caching components, and concentrates the application state within the user agent components.

In this style, software components are recast as network services. Clients request resources from servers (or proxy servers) using the resource's name and location, specified as a Uniform Resource Locator (URL, [25]). All interactions for obtaining a resource's representation are performed through a synchronous request-response pair using an open standard network protocol (HTTP, [68]). Requests could also be relayed via an assortment of proxies and caches, all without changing its semantics.

[12] "Any problem in computer science can be solved by another level of indirection"
— David Wheeler, chief programmer for EDSAC [120]

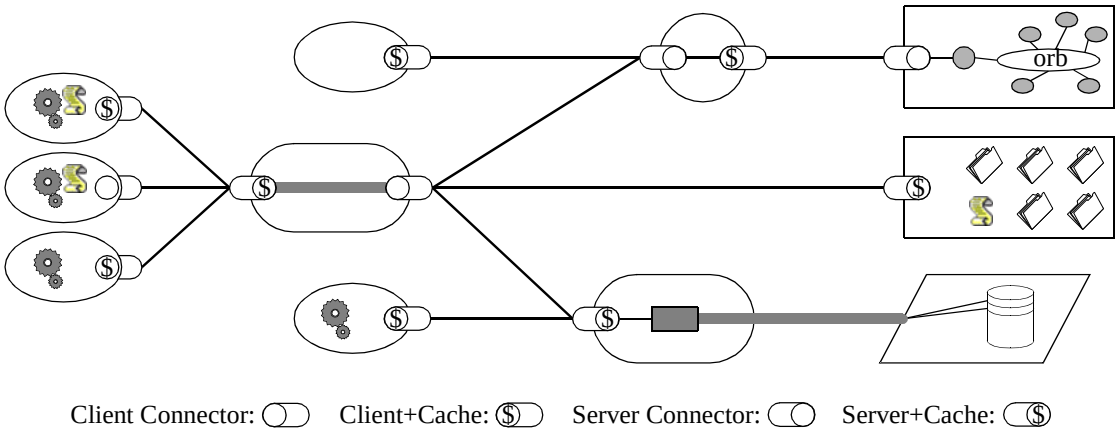

Figure 5-8. REST

Figure 3: The REpresentational State Transfer Style, from [67].

Figure 3 depicts a REST-style architecture that also reveals several of the simpler styles it was derived from. Moving from left to right, we can follow client's requests as they are directed through firewalls, caches, and protocol gateways to a set of three different servers. Along the way, the diagram also intends to show that REST is a stateless, cacheable, layered client-server architectural style that requires uniform interfaces (and can also take advantage of mobile code to extend those interfaces).

On the left-hand side we see three USERAGENTs. two of which are typically browser-like Web clients because they contain a 'history list' for user navigation (indicated by a client-side cache). All of them contain script-language interpreters (indicated by the gears) that enable the so-called 'Code on Demand' architectural style for dynamically extending the range of behaviors implementable on the client.

In between, there are several layers of PROXYSERVERs and GATEWAYs. The first one that all three clients are connected to is a caching proxy. Unlike each browser's private history caches, this sort of shared cache can take advantage of common access patterns across users — as well as enforce organization-wide rules, as a firewall might. It can support multiple inbound connections because the interactions are stateless; it does not matter which user agent is making the request because all session information must be encoded into each request. REST's emphasis on uniform interfaces is also revealed by the three outbound edges: the proxy can connect to servers, proxies, and even gateways to non-Web information systems interchangeably. Furthermore, the fact that one proxy can choose to "chain" to another proxy indicates that REST is a layered style; to the degree that proxy components can transform data, this makes REST a pipe-and-filter style as well.

On the right-hand side, we see several types of ORIGINSERVERs: an application server containing running objects and an Object Request Broker (ORB); a file server containing directories and programs; and a WAIS server containing a repository

database. We see that there is a gateway component between the caching proxy and the WAIS server because it converts the uniform REST interface to an external protocol when accessing WAIS, as depicted by the grey edge. Separately, there is also a cache *within* the file server, which is another form of load-balancing for a server that might physically be implemented on several disks and processors.

As an aside, the reason we choose to step back from this illustration and claim that REST is not only a network-based architectural style, but also an internetworking style itself, is that these derivation arguments apply equally well to IP internetworking. A mesh of IP routers can be considered an "application" for moving information around. The IP standards and host requirements [32] for TCP/IP programs ("stacks") are also stateless and layered client-server style. The converse of this insight is that REST treats software components exactly as IP treats networking devices: the uniform `GET` interface in HTTP is similar to the uniform receive-packet/fragment interface in IP. It is a flexible enough interface to trigger any kind of computation, yet fixed enough to connect to any kind of device/component.

4.1.2 Limitations of the "One-Way Web"

REST's exclusive focus on supporting the specific domain of global hypertext/hypermedia limits the range of applications it can support. To be sure, a vast range of traditional client/server applications have already been successfully 'ported' to REST over the last decade. Applications that rely on real-time or event-based interactions have not.

Returning to the discussion of C2 in §3.1.1, consider how it compares to REST:

> As with other event-based schemes, C2 is nominally push-based, though a C2 architecture could operate in REST's pull style by only emitting a notification upon receipt of a request. However, the C2 style lacks the intermediary-friendly constraints of REST, such as the generic resource interface, guaranteed stateless interactions, and intrinsic support for caching. [67]

Extending REST to support the additional capabilities of C2 or other event-based styles would require significant redesign of REST, but the benefits would be significant. What is at stake is no less than an opportunity to complete the vision of the Web as a *two-way* medium for reading, writing, and *interacting* with the entire "universe of network-accessible information" [20].

Conversely, the challenges facing architects attempting to develop interactive applications in the REST style mainly stem from three key limitations of what we call the "One-Way Web": REST has been designed to support one-shot, one-to-one, and one-way information flow.

One-shot. Every representation transfer must be initiated by a client; generates a single response; and if that response is an error message, there are no normative rules

for reacting to it. Even though the underlying connector technology may rely on a reliable, ordered delivery mechanism, REST's model of network interaction is not very fault-tolerant.

E-mail, on the other hand, is an example of an application and network protocol that work together to specify rather intricate behavior that ensures robustness in the face of message loss, delay, and congestion. According to the Simple Mail Transfer Protocol (SMTP, [182] and [54]), a Mail Transfer Agent (MTA) must be able to guarantee that messages are in stable storage before sending any acknowledgement messages. Furthermore, an MTA must store messages and attempt delivery for several days (unlike the few-minute timeouts for TCP to fail).

To be sure, a few REST error codes suggest actions for the client to take, such as redirects, password challenges, and temporary outage notices, but those are also strongly tied to the hypertext domain and human end-users. Many application architects expect a style that aims to replace the dominant 3-tier client/server style to provide equivalent mechanisms for reliable, ordered invocation of components — REST's send-once-and forget model is not sufficient.

One-to-one. Every representation transfer proceeds from one client to one server, which may optionally forward it along to another server, and so on. There is no alternative to REST's nested "proxy chain" for transferring information to an entire group of components. Even in Figure 3, the branching depicted there is not intended to be simultaneous: each client request message specifies which server to relay it onward to. Furthermore, such instructions for proxy-selection are only "hop-by-hop": no single client is in a position to specify an acyclic graph for information flow.

In the REST style, the only way to extend a system is by interposing a so-called 'active proxy' into the chain [124]. As [67] states, "Within REST, intermediary components can actively transform the content of messages because the messages are self-descriptive and their semantics are visible to intermediaries."

However, a critical limitation for architects of decentralized systems is that a nested pipeline requires every component involved to trust every other component. Intermediaries owned by agencies that do not trust each other must be invoked "serially," by having the client directly call each one in turn with the intermediate results.

Again, application architects expect a style that aims to replace the dominant 3-tier client/server style to provide equivalent mechanisms for group communication and ordered invocation — REST's pipeline model is not sufficient.

One-way. Since every representation transfer must be initiated by the client, and every response must be generated as soon as possible (the statelessness requirement) there is no way for a server to transmit any information to a client asynchronously in REST. Furthermore, there is no direct way to model a peer-to-peer relationship. The

only workaround is to posit paired clients and servers that use a common repository to have one component send and receive data simultaneously using REST.

To be sure, there are certain representation formats that happen to be isochronous. A video representation, for example, could just be a very large file that takes time to download, or it could be a live camera generating the contents in "real-time" and that arrive at the client with roughly the same delays. An even more extreme example is the JavaScript tunneling technique we pioneered in MOD_PUBSUB, for sending real-time events in the guise of a very long, slow, HTML download. In fact, that approach could be considered a weak form of the Code-on-Demand style, which remains REST's most popular escape mechanism for accommodating interactive information (Java applets, Macromedia Flash, etc.).

Given the difficulty of managing any sort of asynchronous data transfer from servers to clients, much less establishing simultaneous agreement, high-frequency resources are rarely used in REST-style applications today. Instant messaging applications are only the most visible example of the sorts of client/server application that still are not implemented inside of Web browsers. Many more custom enterprise applications remain locked into the legacy environment of terminal emulators for lack of a two-way transfer mechanism in REST.

4.2 REPRESENTATIONAL STATE TRANSFER (REST)

Before extending REST to support the "Two-Way Web," we need to specify precisely what it is and what properties it can induce. The goal of this section is twofold: to introduce the aspects of the original REST style we rely on for our subsequent work; and to introduce the structure of the arguments we will advance in our subsequent work.

In the case of REST, we are "working backwards" from a style to a formal model that justifies its behavior: inducing the property of consensus. We will be specifying the abstract capabilities for reading the value of remote variables and ensuring that they are currently valid; defining the new architectural elements that represent those capabilities; validating that these components can fulfill their abstract specifications in practice; and finally, notes on implementing the style.

4.2.1 Property Specification

Centralization is easy to support if a variable can only be assigned a value once. Immutability simplifies the requirement for simultaneity: the value that a follower receives is guaranteed to remain in agreement with the leader's value indefinitely.

References to this sort of variable require *consensus* — there is one decision to be made by the leader, and all others must follow. Consensus, in turn, will be used later

on as a stepping-stone for establishing simultaneous agreement in general, by using synchronized expiration times.

The most direct mechanism for supporting a write-once centralized variable is as simple as ensuring that the leader broadcasts the value it assigned to all possible followers. However, it is not feasible to enumerate 'all possible followers' in any dynamic, loosely-coupled system, because new components can be added at runtime.

We propose an alternative mechanism that is client-initiated: *read()*. This way, new followers can request the value of the centralized variable at any time. The tradeoff is that *read()* also introduces additional delay. While the broadcast solution only would take at most d seconds to deliver a message across the network, *read()* requires an additional factor of d so the client can issue a request first.

Our aim in this subsection is to specify whether this mechanism can enforce the property of consensus. Informally, consensus requires first, that *read()* returns *some* representation (rather than null); and second, that the representation has not expired yet.

First, we need to establish the strongest postcondition of *read()*. Formally, if X is the leader's centralized variable and Y is a follower's variable, and both are represented by functions that vary with time t, then initiating the *read()* procedure at time t = NOW, with random network latencies λ_1 and λ_2 for the messages to and from the central server, is specified by the following Hoare triple:

$$\{\ (0 < \lambda_1 \le d) \wedge (0 < \lambda_2 \le d)\ \}$$
$$Y := read(\text{"X"});$$
$$\{\ Y(\text{NOW} + \lambda_1 + \lambda_2) = X(\lambda_1)\ \}$$

Program 1: Specification of *read()*.

That is, *read()* returns the value of the variable named X at the moment the request arrives at the server. To compute the weakest precondition of this program, we must quantify the latencies λ_1 and λ_2 over the entire interval $(0, d]$:

$$\{\ true\ \}$$
$$Y := read(\text{"X"});$$
$$\{\ Y(\text{NOW} + 2d) \in \forall\ \lambda_1 : 0 < \lambda_1 \le d : \cap\ \{\ X(\text{NOW} + \lambda_1)\}\ \}$$

Program 2: *read()* quantified over possible latencies.

That is, a *read()* can be invoked at any time, but the strongest postcondition it assures is that, after at most $2d$ seconds, the value of Y will become *some* value that X

had during the first d seconds. This reflects a sort of "race condition": the network could deliver the request before any value has been assigned to X at all.

This postcondition for *read()* is not strong enough to *guarantee* consensus, only indicate that it might be possible. For that, its precondition would have to be strengthened to assume that X has already been defined before the request is made, and is never changed again:

$$\{ \forall\, t : t \geq \texttt{NOW} : X(t) = X(\texttt{NOW}) \}$$
$$\quad \Upsilon := \mathit{read}(\texttt{"X"});$$
$$\{ \forall\, t : t \geq (\texttt{NOW} + 2d) : \Upsilon(t) = X(t) \}$$

Program 3: *read()*, if the variable has already been assigned a value.

The proof is straightforward, since the "set of values X had during the first d seconds" has a single member; and we can strengthen the equality of X and Υ to hold indefinitely afterwards.

The next challenge that suggests itself is: What happens if X *is* updated? We can specify expiration times in order to establish consensus over a finite duration instead. However, we will find that this still makes it impossible to guarantee consensus for clients invoking *read()* "too late," within $2d$ of a value's expiration.

We transform the program analyzed above by redefining the value of a variable to be a *(value, interval)* pair. This will require positing a globally synchronized clock, which itself if a solution to the simultaneous agreement problem. Using synchronized clocks to define the *interval*, we can then define a function called *fresh()* that verifies whether a value is still valid before using it. The Hoare triple for the *fresh(Value, Start, Lease)* test specifies that it can be executed at any time:

$$\{ \mathit{true} \}$$
$$\quad \Upsilon := \mathit{fresh}\ (V, S, L);$$
$$\{ S \leq \texttt{NOW} < S{+}L \rightarrow \Upsilon = V$$
$$\ (\texttt{NOW} < S) \vee (\texttt{NOW} \geq S{+}L) \rightarrow \Upsilon = \varnothing \}$$

Program 4: Specification of *fresh()*.

With this in hand, we can now specify the behavior we expect of the REST architectural style:

$$\{\ \forall\, t:\ \ S \le t < S+L : X(t) = (X_1, S, L)$$
$$\quad \wedge\ \ S \le \texttt{NOW} < ((S+L) - 2d)\ \}$$
$$\quad Y := \mathit{fresh}(\mathit{read}(\texttt{"X"}));$$
$$\{\ \forall\, t : (\texttt{NOW} + 2d) \le t < (S+L)\ : Y(t) = X(t) = X_1\ \}$$

Program 5: Composition of *fresh()* and *read()*.

That is to say, if the leader's value is defined, and stays that way for long enough to send a *read()* request and wait for the return, the follower's value will be equal to the leaders.

The proof is straightforward, since the duration of the interval in the postcondition is nonzero as long as the request is made during the value's validity and early enough for *read()* to terminate. Hence, the preconditions NOW ≥ S and NOW<$((S+L) - 2d)$, respectively. Another consequence is that the minimum value of L must be $2d$ for the interval in the postcondition to exist at all.

4.2.2 Style Definition

> The REST architectural style induces (the possibility of) agreement by constraining all components that refer to shared resources to request a current representation from an ORGINSERVER each time.

The REST architectural style meets the specifications given above. It adopts the client-initiated mechanism rather than the broadcast notification mechanism we discussed earlier. This choice enhances its scalability and dynamism, since a subscription model requires additional state to be stored at the server.

As depicted in Figure 4, REST's ORIGINSERVER implements *read()* and the client-side USERAGENT invokes it using the GET method. Our only clarification is to ensure that both implementations of *fresh()* are synchronized with the aid of a GLOBALCLOCK.

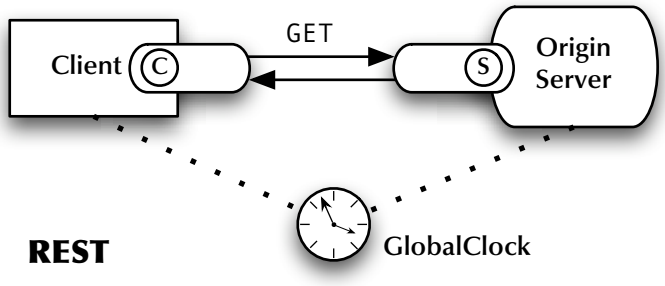

Figure 4: Illustration of the REST architectural style.

Recall that in REST, a resource can have multiple, simultaneously valid representations. In the context of a global hypermedia application, those variants might be in different file formats or different languages. Rather than testing for byte-by-byte equality, the condition is whether a human reader would consider them equivalent.

This level of indirection is the only significant departure from our formal definition of consensus between variables.

4.2.2.1 The GlobalClock Component

Every component must refer to a centralized GlobalClock, or else refer to a local proxy that is synchronized with it. This is how each representation (notification) can refer to a strict lease expiration deadline. Synchronized lease initiation times are also necessary to ensure that a follower is not in disagreement during the transmission of a notification (or takes unfair advantage of advance knowledge of the new value).

Of course, the necessary degree of clock synchronization is determined by domain-specific requirements. We believe a popular technique for breaking ties in leader-election routines for decentralized systems will be to use a combination of component IDs and timestamps. This is only reasonable as long as the probability of collision is kept low enough. Therefore, we note the resolution of a GlobalClock must be much finer than the timescale of any of the phenomena being modeled.

4.2.3 Validation

REST *can* induce consensus for centralized resources, as summarized in Table 3. It cannot *guarantee* consensus because there is no way to be absolutely sure that the leader's value has been set already, and will remain the same after the reply message eventually arrives. Our validation for this claim is an argument that, by construction, the architectural elements and behavioral constraints of REST correctly implement the abstract mechanisms that were already validated in §4.2.1.

	Goal	New Elements	New Constraints	Induced Property
REST	Refer to a centralized resource.	GlobalClock makes explicit how clients, servers, and caches are synchronized.	OriginServer must always specify a consistent expiry deadline if the resource is ever to be updated.	*Consensus:* Ensures that local resource proxies *could* agree with leader's value.

Table 3: Summary of the REST style.

Read() describes an ideal function that returns a representation instantaneously. A REST OrginServer's implementation of GET will be somewhat more complicated,

since representations are generated with reference to external data sources ranging from slow disks to remote service gateways. This induces apotential error of ε, since the reply to a request that arrives at λ may only be generated at λ+ε. One way to accommodate this is to increase *d* by the amount of ε appropriate to the application at hand. That treats representation-generation latency as one more component of overall network latency.

Similarly, *fresh()* describes an ideal function that works with absolute timestamps of arbitrary precision. In practice, the GLOBALCLOCK we added to REST must permit a degree of error in making time comparisons at separate locations, both due to limitations of finite precision and relativistic limits to accuracy. Again, one way to accommodate this timing error is to shorten effective lease durations by an appropriate ε.

A more serious limitation that has gone unstated so far is that *fresh()* depends on a validity *interval* being specified in the original message from the server *a priori*. Many real-world ORIGINSERVERs do not specify when the next permissible resource update is scheduled; the external environment could change it at random (e.g. editing a file by hand on a filesystem-based Web server).

One practical bridge between an implementation of the REST style and our abstract model is to set a default lease and force the server to discard or delay updates that arrive sooner than that lease deadline. This is the *heartbeat model*, which we will discuss further in the next section, where we will extend REST to guarantee consensus, and indeed, even a limited form of simultaneous agreement as well.

4.2.4 Implementation Issues

We would like to note two potential implementation issues facing developers of REST-style infrastructure: lease expiration policy and clock synchronization.

4.2.4.1 Expiration

Moving from the abstraction of ORIGINSERVER to practical implementation, we need to explain how leases are determined (the terms *S* and *L* in the specification section). Its response to a GET request must include not only the current representation of the resource, but also the time it is generated and the time it will expire.

If the expiration time is not specified, its default value must be ∞. The client simply has no other knowledge of when it might expire, which forbids the server from ever changing it again. This is true even if the deadline is later determined some time after the representation is first generated — if any client received an indefinite lease, every future client will be so bound as well.

There are two practical solutions to this risk. The first is to actually use write-once resources: by putting timestamps into the *name* of a resource and redirecting requests for the generic resource to the time-specific ones, each of those will only

have to be assigned once. This pattern is often found in online newspapers and Weblogs.

The second solution is for the server to consistently reply with expiry deadlines of NOW+d+ε as long as no future update has been scheduled yet. On the Internet, d is often assumed to be roughly five minutes, per the 255 second IP packet lifetime or the TCP TIME_WAIT delay. In this case, every requesting client is still able to "use" the reply value for ε. As soon as the server wishes to update a resource, it freezes the update deadline — so requests after this point may fail due to the vagaries of network latency — and then puts the new value into effect after that.

4.2.4.2 Synchronization

Moving from the abstraction of GLOBALCLOCK to practical implementation, we need to note two additional concerns: clock skew and authenticity. These are very real problems on the Web today [158].

Technically, "skew" refers to two different sources of timing error: initial synchronization and errors in clock rates ("drift"). Regardless, what is critical is that the amount of the error must be much less than the minimum useful life of a representation, $(L_{min} - d)$. The Simple Network Time Protocol (SNTP, [154]) shows that it is feasible to limit drift in proportion to the variance of the average latency between trusted timekeepers (rather than the maximum latency), which is on the order of milliseconds across the modern Internet [64]. Furthermore, the US Global Positioning System (GPS) is a practical demonstration of how closely clocks can be synchronized between fixed positions in an inertial frame — within nanoseconds.

This establishes that GLOBALCLOCK can be established to the requisite level of precision, but accuracy is another matter entirely. Then, the key is deciding which timekeeping services an architect trusts for a particular application. GPS suffices if one trusts the US Government; otherwise the former Soviet Union's GLONASS and forthcoming European Union Galileo systems serve as sovereign alternatives to American military control of so-called 'selective availability' accuracy limits [3, 33]. Cryptographically-secure network notary services may also prove necessary for clock synchronization across agency boundaries [103].

4.3 REST WITH POLLING (REST+P)

In this section, we will present a way to use REST that guarantees consensus, and even manages a degree of simultaneous agreement. The maximum possible update rate for centralized resources within the REST architectural style will be shown to be $1/3d$ — we will have to derive entirely new styles to actually achieve the theoretical maximum of $1/d$ in the next chapter.

4.3.1 Property Specification

To guarantee consensus, we need merely to eliminate the conditions that prevented it. We found two such conditions in our analysis of REST: that the value of X has not been defined yet, or that X's value will expire before delivery. Our solution is *poll()*, a new program that guarantees consensus. Its specification is merely to keep re-reading until Υ is defined:

$$\{\ \forall\ t:\ \ S \le t < S+L : X(t) = (X_1, S, L)$$
$$\quad \wedge\ \ \texttt{NOW} + 2d < (S+L)\ \}$$
$$\quad *[\ \Upsilon = \varnothing \rightarrow \Upsilon\ := \mathit{fresh}(\mathit{read}(\texttt{"X"}));\ ]$$
$$\{\ \forall\ t : (\max(\texttt{NOW}, S + d) + 2d) \le t < (S+L)\ :\ \Upsilon(t) = X(t) = X_1\ \}$$

Program 6: Specification of *poll()*.

Unlike the specification at the end of §4.2.1, *poll()* can begin executing even before S occurs. In return, *poll()* guarantees consensus as long as whatever value X is assigned to, remains valid for more than three network delays. It will take no longer than two network delays after it begins execution, or three network delays after X is assigned a value, whichever comes later.

The base case is that $S \le$ NOW, in which case *poll()* executes once, and consensus is established no later than NOW + $2d$, as in §4.2.1. The inductive step is to repeat a new execution of *read*(), given that the current one failed.

Consider a failure case. Returning $\varnothing$ implies that that $S >$ NOW $+ \lambda_1$. Furthermore, the scheduled time of the next execution of the loop is defined as NOW $+ \lambda_1 + \lambda_2$; call this NEXT_RUN. Combining the two, we can infer that $S >$ (NEXT_RUN $- \lambda_2$). Since the maximum value of λ_2 is d, we can further establish that $S >$ (NEXT_RUN $- d$).

Since the clock advances on each execution (NOW := NEXT_RUN, as the bookkeeping would have it), failure adds the critical piece of knowledge in the inductive step: $S >$ NOW $- d$. This again leaves us with two cases: either $S \le$ NOW or S > NOW. In the latter case, the monotonic increase of NOW allows us to deduce that, at some point, $S \le$ NOW, reducing the problem to the former case.

Thus, executing *read()* repeatedly must succeed eventually. Once it does, that final execution of *read*() will terminate by NOW $+ 2d$ — but since we know $S >$ NOW$-d$, that deadline is no later than $S+3d$. Combined with the base case, the entire program will establish consensus by max(NOW$+2d$, $S+3d$). Finally, we factored out $2d$ to state the postcondition used in the Hoare triple above.

All that remains is to prove that the interval in the postcondition is nonempty, as long as the minimum lease duration is $3d$:

$$
\begin{aligned}
& \exists\, t : (\max(\texttt{NOW},\, S + d) + 2d) \le t < (S+L) \\
\Rightarrow \\
& \max(\texttt{NOW},\, S + d) + 2d < (S+L) \\
\Rightarrow \\
& L_{min} = \max(\texttt{NOW},\, S + d) + 2d - S \\
\Rightarrow \\
& \texttt{NOW} \le S+d \rightarrow L_{min} = 3d \\
& \texttt{NOW} > S+d \rightarrow L_{min} = 2d + (\texttt{NOW} - S)
\end{aligned}
$$

That is, if *poll()* starts no later than d after X has been assigned, X need only hold for $3d$. However, if we can assume X can't stay $\varnothing$ indefinitely — that it is always assigned *some* value eventually — then we can establish $L_{min} = 3d$.

This follows from our induction above: if X is defined, but the remaining lease lifetime is too short, the first pass through the *poll()* loop will fail. If we assume that X will eventually be assigned again, then we know that at some point, the loop *will* terminate — and termination implies $S > \texttt{NOW} - d$. Substituting that into the expression just given, $2d + (\texttt{NOW} - S)$, and we find that $L_{min} = 3d$ in both cases.

In fact, if the loop above were suitably modified to restart the loop every time Υ's value became stale again, we could even establish simultaneous agreement:

```
{ L > 3d }
    *[  *[ Υ = ∅ → Υ    := fresh(read("X")) ];
        *[ Υ ≠ ∅ → Υ    := fresh(Υ) ] ]
{ ∀ t : t ≥ NOW + 2d:  Υ(t) = (X(t) ∨ ∅) }
```

Program 7: Repeating *poll()* when values expire.

A formal proof would follow along the lines we will discuss later, in §5.1.1. An informal argument is that since the value of X alternates between defined and undefined values, the postcondition is trivially true when it's defined, and Υ will follow each assignment after a delay of at most $3d$. Hence, we describe *poll()* as capable of "slow" simultaneous agreement of at most $\frac{1}{3d}$ Hz.

Afterword: wait(). To be sure, it is also possible to describe a function very similar to *read()* that would at least reduce the factor of $3d$ in *poll()* to only $2d$. *Wait()* is a semi-stateful variant of *read()* that delays returning until X is defined. Formally, it can be described as follows (assuming $X := X_1$ at time T):

$$
\begin{aligned}
&\{\ (\circ < \lambda_1 \le d) \wedge (\circ < \lambda_2 \le d) \\
&\wedge \exists\, T:\quad (\forall\, t : t < T:\ X(t) = \varnothing\ ) \\
&\qquad\quad \wedge\ (\forall\, t : t \ge T:\ X(t) = X_1\ )\} \\
&\quad \Upsilon\ := \mathit{wait}(\texttt{"X"}); \\
&\{\ \Upsilon(\max(\texttt{NOW}+\lambda_1,\ T) + \lambda_2) = X(T)\ \}
\end{aligned}
$$

Program 8: Specification of *wait()*.

Or, after eliminating λ by quantification, $\Upsilon(\max(\texttt{NOW}+d,\ T) + d) = X(T)$. If T occurs before the request arrives at the server, then after $2d$, Υ will be assigned that value. If T is further into the future, however, then the reply message is sent precisely at T and Υ will be assigned that value no later than $T+d$. In other words, *wait()* behaves exactly like *read(),* except that it delays returning until a value is assigned. A version of *poll()* that used the *wait()* primitive could achieve a maximum update frequency of $^1/_{2d}$.

4.3.2 Style Definition

> The REST+P architectural style induces agreement by constraining all components that refer to shared resources to repeatedly re-request a current representation from an ORGINSERVER at all times.

The REST+Polling architectural style meets the specifications given above. It adopts a client-initiated continuous polling mechanism rather than the one-shot query model of REST alone. Unlike the new styles we will derive in the next chapter, REST+P has the substantial virtue of not modifying the definition of a server component, only changing the behavior of clients.

As depicted in Figure 5, all REST+P needs is a modified POLLINGCLIENT, whose repeated GET requests are represented by multiple arrows.

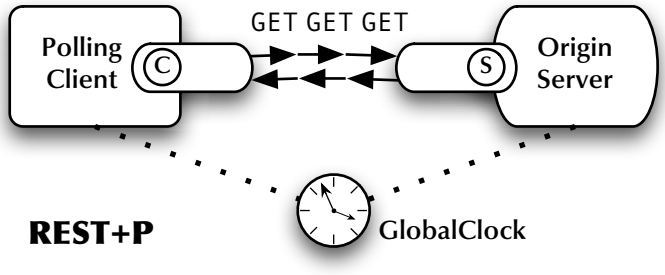

Figure 5: Illustration of the REST+P architectural style.

4.3.2.1 The PollingClient Component

A PollingClient will issue a GET request to an OriginServer and continue to re-issue that request until the response becomes a valid representation. It must re-issue requests no later than d seconds apart.

Requests into a PollingClient component will block until some content can be returned. An alternative event-based interface is for a request into a PollingClient component to return a series of representations over time by re-starting its polling loop whenever the current representation expires.

4.3.3 Validation

As summarized in Table 4, continuous client-initiated polling (REST+P) can guarantee consensus and even induce simultaneous agreement, though only for centralized resources that change no more frequently than $\frac{1}{3d}$.

	Goal	New Elements	New Constraints	Induced Property
REST+P	Refer to a *mutable* centralized resource.	PollingClient that reissues a new GET immediately upon each expiration.	Polling request rate must exceed $\frac{1}{2d}$ Hz.	*"Slow" Simultaneous Agreement:* Local proxies *will* agree with leader if its update rate is less than $\frac{1}{3d}$ Hz.

Table 4: Summary of the REST+P style.

Our validation for this claim is an argument that, by construction, the architectural elements and behavioral constraints of REST+P implement the abstract mechanisms that were already validated in §4.3.1.

Poll() describes an ideal program that reissues a request instantaneously. At worst, read requests must not be longer than $2d$ apart, or the proof fails. Since a REST+P PollingClient implementation could be quite inefficient if it re-requested continuously, the scarcity of bandwidth or processing power could delay requests by an additional factor of ε. One way to accommodate this is to increase d by the amount of ε. A more involved formal argument could be advanced to tighten this practical bound to $L_{min} = request\_interval_{max} + d$.

4.3.4 Implementation Issues

The polling loop is a popular technique for increasing interactivity without burdening server implementation — when the tradeoff between the benefits of state lessness and the increased overhead of processing additional requests justifies it. Of course, polling also places a burden on client implementations to retry requests, but a fundamental assumption of REST (and arguably of Internet-scale architectures in

general) is that "nearby" computing resources are faster to access and more abundant, thus favoring processing at the edges of a network.

Indeed, today's Web browsers already include vestigial support for this mode of interaction. In HTML, a `<META HTTP-EQUIV="REFRESH" CONTENT="N">` element in the header of a document indicates that page should be reloaded every N seconds.[13]

As mentioned earlier, unregulated polling can waste bandwidth and processing power. A PollingClient should not issue more than one pending GET request at a time.

Furthermore, an additional optimization that leases enable is suppressing polling while the current representation is still fresh. A subsequent *read()* request could be issued only at the time of expiry without invalidating our performance analysis, since a reply only takes at most $2d$, while $L_{min} > 3d$. An even more efficient implementation could opt to wait until the full $2d$ elapses between requests, without loss of generality.

[13] Despite appearances, though, this HTML header does not imply that there is a "refresh" facility in HTTP/1.1 *per se*; at most, the server can issue an advisory error message to retry the failed transaction after a specified interval.

Chapter 5: Centralized Systems

Client-initiated polling approaches are not sufficient for maintaining simultaneous agreement between a centralized resource and a local proxy. Instead, we propose an event-based approach that permits the central resource to broadcast notifications of its state changes. Our insight is to recast the concept of a resource in REST into an event source that emits event notifications corresponding to each change in its representation(s). Building upon this reinterpretation of the REST architectural style, we can induce several distinct properties: simultaneous agreement using events; multilateral extensibility using routes; and simultaneous invocation by using publish-and-subscribe integration.

5.1 Asynchronous REST (A+REST)

In this section, we will present a way to extend REST into an event-based architectural style. By taking advantage of asynchronous notification, followers can be updated within at most d seconds of a change in the leader's value.

5.1.1 Property Specification

We need a different sort of mechanism to induce simultaneous agreement for a mutable centralized resource that actually permits the maximum update rate of $^1/_d$ that we derived in §2.3.1. We propose adding a broadcast mechanism to extend *read()*, called *subscribe()*, that we can prove is correct and minimizes total latency.

First, though, we need to define *notify()*, a new operation that lets the leader directly set the value of a follower's variable. This program could be described as an "instantaneous *read()* request," since it eliminates the latency of sending the initial request. Formally, its specification is:

$\{ 0 < \lambda \leq d \}$
$\quad Y := notify(\text{"X"});$
$\{ Y(\text{NOW} + \lambda) = X(\text{NOW}) \}$

Program 9: Specification of *notify()*.

Or, quantifying the random value λ over its entire interval $(0, d]$ to relax the precondition so that it can be invoked at any time:

{ *true* }
    Υ := *notify*("X");
{ Υ(NOW + *d*) = *X*(NOW) }

Program 10: *Notify(),* quantified over possible latencies.

In conjunction with *fresh()*, we can assign the value of the local variable for the duration of the remaining lease interval, if any:

{ $\forall t : S \le t < S+L : X(t) = (X_I, S, L)$ }
    Υ := *fresh*(*notify*("X"));
{ $\forall t : (\text{NOW} + d) \le t < (S+L) : \Upsilon(t) = X(t) = X_I$ }

Program 11: Composition of *fresh()* and *notify()*.

This program, in turn, is used in conjunction with another program running at the centralized server that invokes *notify()* every time the value of *X* changes. To describe the behavior of *subscribe()*, we need to introduce CLOCK, a monotonically increasing value representing the passage of time:

{ $L > d$ }
    X_{Last} := $\varnothing$;
    *[$X(\text{CLOCK}) \neq X_{Last} \rightarrow X_{Last} := X(\text{CLOCK})$;
                Υ := *notify*(X_{last}, CLOCK, (CLOCK + L));]
{ $\Upsilon(t) = (X(t) \vee \varnothing)$ }

Program 12: Specification of *subscribe()*.

That is, *subscribe()* establishes simultaneous agreement between *X* and Υ indefinitely. Our argument proceeds by enumerating the value of the function *X(t)* as a series of disjoint *(value, interval)* pairs. A lease interval, in turn, is a *(start, lease)* pair such that $lease \ge L_{min}$. We must show that this program sets the value of any follower variable $\Upsilon(t)$ to the series of corresponding disjoint pairs *(value, (start+d, start+lease))*. Since $L_{min} > d$, each interval is nonempty, so Υ will remain in simultaneous agreement with *X*.

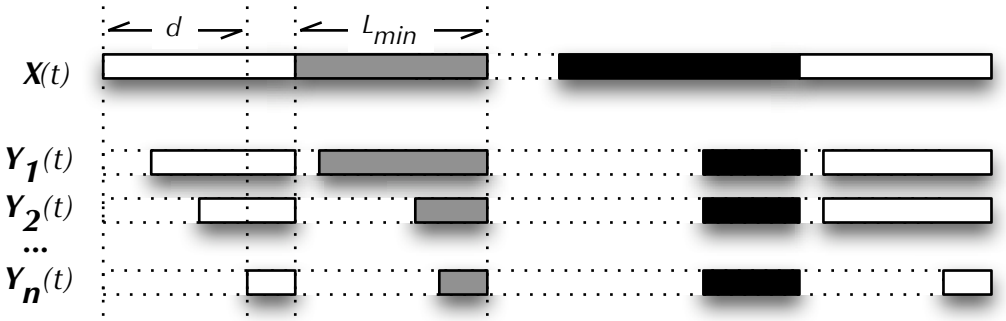

Figure 6: Illustrating simultaneous agreement using notifications with leases.

Figure 6 visualizes our argument. As X is assigned a series of colors, each follower Υ_1, Υ_2, ..., Υ_n receives each message after some random latency λ. The worst-case intersection is for some Υ_n that always encounters λ=d.

Returning to our formal argument, we must establish the base case, that simultaneous agreement holds initially; and the inductive step that maintains it. Initially, Υ(NOW) is presumed to be $\varnothing$, so the condition holds. The only step that can modify the value of Υ is *notify()*, which also preserves simultaneous agreement during the interval [CLOCK+d, CLOCK+L). And finally, by our definition of *X(t)* as a series of disjoint leased values — every successive pair in the series must have a distinct *value* — each change to X will falsify the guard of the loop, triggering a notification. Furthermore, as long as it takes less than d seconds to execute *notify(),* the busywaiting loop is guaranteed to detect *every* change in the value of X, and hence Υ will also be set to each value assigned to X after *subscribe()* is invoked.

Expiration Policy. What happens when *lease* is not known at *start*? This is a typical case, where the timing of the next update is unknown. The solution will be either to use a heartbeat — broadcasting a new event notification every L_{min} seconds — or to send explicit invalidation messages — delaying an update by d seconds to send a "clear" message to every follower before switching over to the new value. The optimal strategy will depend on the ratio of the distribution of random update intervals to L_{min}, and the ratio of L_{min} to d.

To illustrate the tradeoff between availability and responsiveness, consider an event source that changes hourly, on average, using a network with a maximum latency of five minutes. An example of a heartbeat solution would be to broadcast the current state every ten minutes, so that all subscribers could be in simultaneous agreement $1 - {}^{d}/_{L_{min}}$ of the time, or at least 50%. The invalidation solution would be to broadcast the new value whenever the source changes, but with a five-minute embargo: the actual changing of the value has to be rescheduled far enough into the future so that every follower can be notified beforehand. This ensures simultaneous agreement, on average, $1 - {}^{d}/_{L_{avg}}$ of the time, or about 92% in this example.

Before issuing a blanket recommendation in favor of invalidation, however, consider what happens if the event source changes as soon as six minutes later. The heartbeat solution cannot re-establish simultaneous agreement for nine minutes, while the invalidation would only take only five.

Conversely, if the next change occurred after nine minutes instead, the invalidation solution would still take five minutes, but the heartbeat would only take one minute — expiry would be already have been scheduled for the tenth minute. Another way of explaining this effect is that, even though the update frequency is limited by *d* in either case, designers need to account for aliasing errors caused by phase-alignment.

5.1.2 Style Definition

> The A+REST architectural styles induces simultaneous agreement by constraining NOTIFYINGORIGINSERVER components to return a continuous stream of updated representations of a resource until a WATCH request expires.

The mechanisms we have been describing so far could be said described as *event-based*. By simply recasting a variable from a time-variate function into an event source — a named object of interest that can be observed for notification of its changes — we can take advantage of several popular architectural abstractions. The key semantic shift this implies is that successive event notifications must have some relationship over time: the values assigned to such a variable must be of the same type. Such time-series of successive values may even permit higher-order analysis such as trend lines or summation.

To extend REST to incorporate the *subscribe()* mechanism proposed above, then, we need a working definition for events that maps onto existing REST elements. Rosenblum & Wolf [197] propose that "An *event* is the effect of the *termination* of an *invocation* of an *operation* on some *object of interest*...Events occur regardless of whether or not they're observed." Our stance is perhaps the opposite, insofar we consider that the very act of 'observation' to be what distinguishes events from messages. Specifically, our concern stems from the realization that 'on the wire,' there is little discernable difference between a messaging protocol and an event-notification protocol. However, there is a dramatic difference between the programming model for a batch message queue and an event handler. Thus, our view might be summarized as "*event notifications* are *messages* that cause *actions*."

In either case, though, there is a clear distinction between the occurrence of an abstract event and the concrete notification of an observation of one[14]. We see in that distinction a clear analogy to the membership function in REST that maps abstract resources to concrete representations.

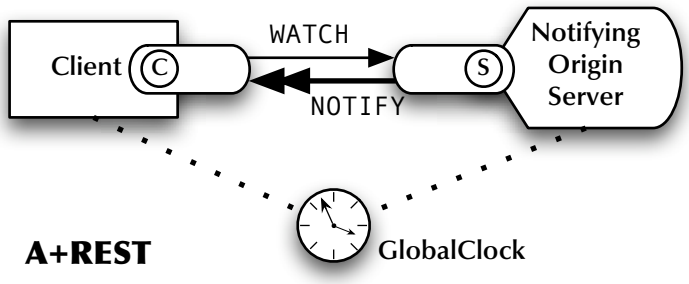

Figure 7: Illustration of the A+REST architectural style.

We propose, then, to recast a REST resource as an event source; and a REST representation as an event notification. The capability REST is missing, though, is asynchronous notification. One does not `GET` the state of an event source; one needs to establish a persistent relationship in order to `WATCH` it. Thus, an Asynchronous REST (A+REST) needs a new element, NotifyingOriginServer, to generate successive notifications of changes to a resource, as shown in Figure 7.

5.1.2.1 The NotifyingOriginServer Component

An NotifyingOriginServer will transfer representations of *every* change to a resource as long as a client stays connected. It could be implemented as a wrapper around an ordinary OriginServer resource by polling it, in turn; or by monitoring an internal resource abstraction such as the filesystem. In either case, an NotifyingOriginServer's connection back to a client must accommodate not one, but a series of response messages for each request.

This requires maintaining state for each subscription relationship. Unlike a `GET`, a `WATCH` request requires specifying the duration of a subscription relationship. In order to support reflection over such long-lived subscription records, we believe each subscription should be a REST resource itself. This will let us 'inherit' a many capabilities from the REST OriginServer model: controlled access to event sources and cryptographic security mechanisms for representation transfers; firewall traversal via proxies; content-negotiation over formats and languages; and caching.

[14] We are using the terms 'occurrence' and 'observation' according to their definition in the event lifecycle model of [196].

5.1.3 Validation

A+REST indeed induces simultaneous agreement for centralized, mutable resources, as summarized in Table 5. We validate this claim by arguing that, by construction, the architectural elements and behavioral constraints of A+REST implement the abstract mechanisms already validated in §5.1.1.

	Goal	New Elements	New Constraints	Induced Property
A+REST	Refer to a mutable centralized resource.	NotifyingOriginServer that can send multiple responses to a WATCH request.	Every resource update must lead to transmission of a new representation to all watchers.	*Simultaneous Agreement:* Ensures that local resource proxies *will* agree with leader's value, even if it is being updated at $\frac{1}{d}$ Hz.

Table 5: Summary of the A+REST style.

Notify() is implemented simply by using REST representation transfers. Since the reply message in an REST request-response pair is itself a representation, complete with caching information, *notify()* can be implemented by transmitting multiple responses while verifying the *fresh()* test for each one using a GlobalClock. To be sure, generating a concrete response message can take time, depending on the representation media type and other factors. This increases d by an ε appropriate to the application at hand. In extreme cases, though, beware that the sheer size of a representation can exceed available bandwidth to the degree that it significantly distorts d.

It can be more difficult to establish whether an NotifyingOriginServer implements *Subscribe()* correctly. It requires each and every change in the value (representation) of a resource to trigger an event notification. A busy-waiting loop needs to run often enough to eliminate the possibility of a 'lost update.' This requires the 'sampling rate' for observing an event source to exceed the maximum possible frequency of the source. Therefore, an NotifyingOriginServer implementation must assure that it has enough computing capacity, etc. to ensure that a scan takes $<< d$ seconds.

5.1.4 Implementation Issues

Moving from the abstract model of an NotifyingOriginServer to practical implementations, we need to consider three key issues: how subscriptions are created and managed; how its connectors interact with the network; and how event sources are observed.

A detour to examine an existing Web-based event notification system is an instructive counterexample. The Jabber instant messaging project [152, 153] includes a browser-based "buddy list" user interface. It implements the *wait()* mechanism discussed in §4.3 using a modified HTTP server to handle requests for an invisible presence icon. Such requests block until some buddy's presence information changes — and then the server drops the TCP connection. Typical Web browsers support scripting language events triggered by a failed image load, which in turn are used to reload the entire buddy list page.

While approach does indeed add a degree of asynchrony to REST model, it is not an NotifyingOriginServer because 1) can only transmit a single notification; 2) its subscription model depends intimately on TCP; 3) and it does not have a robust enough timing model for observing event sources and leasing event notifications to establish simultaneous agreement.

First, support for multiple responses requires modifying the REST client and server connectors slightly, in order to hold open a network connection — similar to REST's existing support for request pipelining and HTTP/1.1 persistent connections [157]. Not only does client-initiation add latency for "re-subscribing" after every buddy list update, the network-layer overhead of re-establishing TCP sessions can cost additional round-trips. Our later work on so-called "isochronous persistent connections" in mod_pubsub addressed these challenges within the limitations of current REST infrastructure by recasting many response messages as a single, slow, very long-lived, representation transfer. It permits full $1/d$ Hz update rates because it eliminates the page reloading round trip as well.

Second, truly simultaneous agreement requires a robust timing model: both to synchronize components and to ensure that event sources are observed on schedule. The Jabber approach still permits arbitrage between followers acting on inconsistent presence information. It neither specifies a deadline by which all followers will stop displaying outdated presence information for a given buddy (heartbeat); nor accommodates pre-caching of information that is scheduled to become valid later on (invalidation). A closely related challenge is scheduling event source observations. In mod_pubsub's Perl Common Gateway Interface (CGI, [102]) implementation, for example, the smallest meaningful, reliable time unit is an entire second, because its key observation loop scans directories on disk for the appearance of new files — which is limited to the one-second resolution of UNIX filesystem timestamps.

5.2 Routed REST (R+REST)

While A+REST tackled the essential challenge of latency, this section will focus on improving REST's support for multiple agencies. Just as asynchrony relaxed the requirement of only one response per request, message routing will relax the requirement that responses must only be sent back to the requester. With redirection

of replies, services can be composed in a manner that eliminates unjustified trust relationships — and minimizes total latency as well.

5.2.1 Property Specification

We would like to define an architectural style that ensures a new property: *multilateral extensibility*. The test of multilateral extensibility is whether new application functionality can be added by using architectural elements owned by several different agencies, all at the same time. Examples of such behavior range from supplying external data files, as when purchasing a third-party font for a word processor; to linking in external programs, as with image processing plug-ins for a photo editor; or even synchronizing customer addresses between billing and marketing systems, as with database synchronization tools in an application integration suite. Since the web of trust between agencies is a *directed* graph, inducing the property of multilateral extensibility requires architectural configurations that enforce directed trust relationships to ensure that data flows only between trusted parties.

Recalling the terminology of owned functions, owned variables, trust relationships, and trusted invocations from §2.3.3, multilateral extensibility can formally be defined as the ability to compose trusted invocations *without* requiring the agencies owning those functions to trust each other.

Consider the application of two agencies' functions to a variable owned by a third: $F_A(G_B(X_C))$. Clearly, a user *(U)* that evaluates such an expression must trust each of the three participating agencies. Formally, *U*'s web of trust must include those edges: $T_U \supseteq \{ U \rightarrow A, U \rightarrow B, U \rightarrow C \}$.

If we evaluate the nested expression directly, though, we find that we need different trust relationships:

$\{ T_U \supseteq \{ U \rightarrow A, A \rightarrow B, B \rightarrow C \} \}$
$\quad y := F_A(G_B(X_C))$;
$\{ y = F_A(G_B(X_C)) \}$

Program 13: Nested composition requires transitive trust.

By our definition of trusted invocations, the owner of a function must trust the owners of all of its arguments, hence the graph given in the precondition. Whereas if we unrolled the expression into three sequential invocations, the owners of those new functions would *not* have to trust each other:

$\{ T_U \supseteq \{ U \rightarrow A, U \rightarrow B, U \rightarrow C \} \}$

$y_1 := X_C$;
$y_2 := G_B(y_1)$;
$y_3 := F_A(y_2)$;

$\{ y_3 = F_A(G_B(X_C)) \}$

Program 14: Sequential composition only requires direct trust.

Recall that assignment only requires the agency on the left hand side to trust the one on the right. Because the result of each evaluation is assigned to a temporary variable owned by *U*, our precondition only specifies a web of trust *(T)* containing edges from *U*.

To resolve this discrepancy, we introduce *delegate()*, a helper function that simply relabels the ownership of information:

$\{ T_U \supseteq \{ U \rightarrow A \} \}$
$y := X_A$;
$\{ y_U = X_A \}$

Program 15: Specification of *delegate()*.

Then, the property of multilateral extensibility can be enforced by simply interleaving calls to delegate between any two agencies that don't trust each other:

$\{ T_U \supseteq \{ U \rightarrow A, U \rightarrow B, U \rightarrow C \} \}$
$y := F_A(delegate_U(G_B(delegate_U(X_C))))$
$\{ y = F_A(G_B(X_C)) \}$

Program 16: Nested composition of *delegate()* only requires direct trust.

In essence, the *delegate()* function "launders" trust relationships by constructing inference chains that are always rooted at *U*.

5.2.2 Style Definition

The R+REST architectural styles induces multilateral extensibility by eliminating the constraint that response messages must return to the ORIGINCLIENT; instead, a ROUTINGPROXY may relay the response to a 3rd party.

Though *delegate()* may appear to be a trivial bookkeeping detail, managing to add it to REST without encountering the unreasonable latency of calling each function sequentially will lead to message Routing (R+REST).

Consider the printing example illustrated in Figure 8-A. In the top row, we begin with an ordinary REST printing scenario. The printer is the resource and `POST`ing a print job to it yields a representation containing the number of pages actually printed. Note that while `GET` requests are required to be idempotent, we are using `POST` for a resource with side effects (in this case, printing).

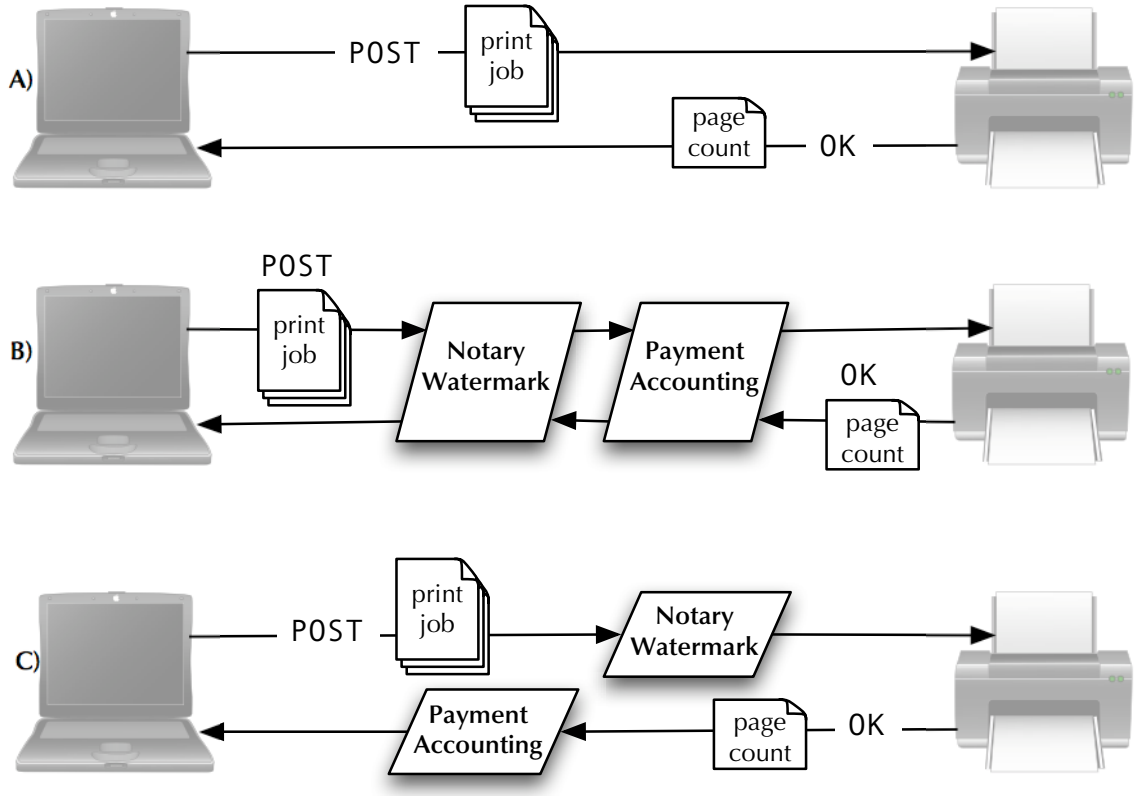

Figure 8: Multilateral extensibility requires eliminating untrusted links. [Note that while row B has six arrows/network delays, row C only uses four.]

Now suppose an application architect is expected to enhance the system with two new features. The agency that owns the printer wishes to account for all pages printed and charge payment from users. At the same time, the user wants to have all of her printouts digitally notarized, with a cryptographic seal encoded as a watermark in the printout.

The REST architectural style makes it comparatively easy to enact these extensions. Because of its emphasis on generic interfaces, proxy servers can be inserted into the path to modify each transaction. The resulting "proxy chain" is depicted in the middle row, Figure 8-B.

However, extending the system *correctly* requires that watermarking be the last step before printing, in case there are other modules added later on (such as, say, converting color to black-and-white). Correctness also requires that the accounting module be the first step that processes the page count "receipt." In other words, the user doesn't want anyone tampering with her notarized printouts, and the owner doesn't want anyone tampering with the page counts.

This uncovers three problems with the REST approach: unjustified trust relationships, increased latency, and hop-by-hop proxy selection.

- Proxies permit an agency to modify both incoming and outgoing representations. Nesting requires complete mutual trust between *every* participating agency, because any proxy along the pipeline could compromise the data flow.
- Conversely, proxies that do *not* modify incoming or outgoing data flow only serve to increase total latency. In an ideal REST solution, WATERMARK is trustworthy because it doesn't care about the reply message, and ACCOUNTING is trustworthy because it doesn't touch the request message. Nonetheless, each of those no-ops still adds *d* to the total latency of the system.
- REST only permits a USERAGENT to select between a server or a proxy server; it does not permit specification of a multiple-hop, end-to-end path. Even extending the application as shown in Figure 8-B still requires the user to explicitly redirect its requests from the printer to WATERMARK; and also requires an entirely out-of-band mechanism for the printer owner to control the WATERMARK server to redirect its output to ACCOUNTING.

The bottom row, Figure 8-C, depicts a better solution. Rather than modeling WATERMARK as a proxy that makes a nested call to ACCOUNTING in order to reply to the user, we model it as an ordinary server that redirects its output to the printer, rather than the original caller. Similarly, ACCOUNTING is only inserted in the data flow at one location this way. Furthermore, with the ability to specify an entire routing path, it is possible for the user to explicitly request that the print job be notarized, printed, and paid for without positing further out-of-band mechanisms.

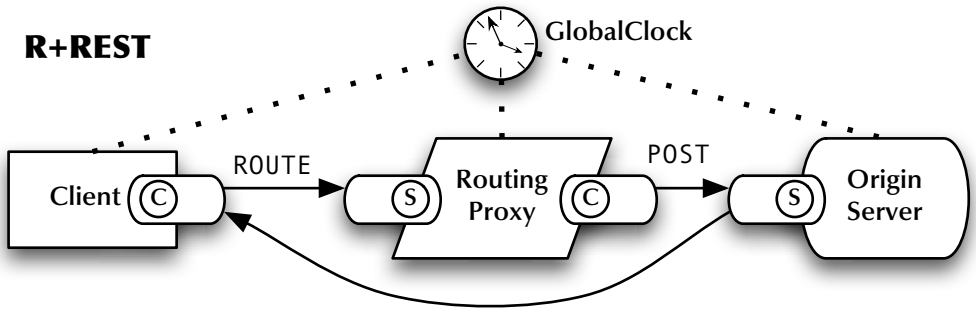

Figure 9: Illustration of the R+REST architectural style.

This calls for a new component that is a cross between an OriginServer and a ProxyServer that can redirect its output to another server, rather than returning to the calling client. Figure 9 illustrates how a RoutingProxy redirects what normally would be a representation transfer back to a client connector into an relayed POST to another server component, as specified by the ROUTE method.

5.2.2.1 The *RoutingProxy* Component

A RoutingProxy component behaves both like a server, since it generates responses without any further nested transactions to other components; and like a proxy, insofar as it relays data onward to other servers. It requires an additional control facility (header) to let the client specify that onward relay path. Of course, in the process of dispatching the request message, the RoutingProxy might itself modify that relay path; that is an accepted consequence of trusting a RoutingProxy rather than calling several services sequentially. There are practical and essential challenges to implementing this facility.

In practice, the initial client has to be able to specify not only the resource identifiers of the onward destinations, but may need to include authentication data for each, and even pass parameters. If the user wishes to browse a bookseller's site with the prices automatically converted to £ by his credit-card processor, the instructions to the bookseller's RoutingProxy would also include credentials for his bank and information about which currency pair he wanted to use. If the bank used a challenge-response protocol (such as digest-authentication, [77]), the interaction model would even require additional round trips. Error reporting in such cases would also be complicated by the need to indicate where along the path such errors occurred, and their severity.

The essential challenge is preventing routing loops. If a path is dynamically constructed, by permitting intermediate RoutingProxy nodes to edit the path and add destinations, then there is a possibility of infinite recursion.

In HTTP, loop suppression is the role of the Via: header. Even if a server could detect all the possible synonyms for its own resource identifiers — an incomputable task because of redirection, DNS aliasing, and other factors — it may prevent a legitimate, layered, reuse of the same service (e.g. nested digital signatures). On the other hand, it is also meaningless to re-sign a message that has changed only trivially (an updated Date: header, say).

The necessary mechanism is one that correctly prevents relaying only of *essentially identical* representations. In REST, that is the role of unique representation entity tags (ETags). Originally developed for cache revalidation, they account for domain-specific rules about what constitutes "essential identity." Thus, a RoutingProxy must maintain a stateful cache of the ETags of message it is currently relaying to ensure that it only relays a request once.

5.2.3 Validation

R+REST indeed induces multilateral extensibility to compose several independently-owned services without presuming trust between those services' owners, as summarized in Table 6. We validate this claim by arguing that, by construction, the architectural elements and behavioral constraints of R+REST implement the abstract *delegate()* mechanism we already validated in §5.2.1.

Our obligation to provide an implementation of *delegate()* is solved completely by backhaul: directly calling the inner component and then subsequently calling the outer component. Our goal is to minimize latency by avoiding an extra network delay to return intermediate representations back to the original caller. A RoutingProxy can correctly implement *delegate()* because the instructions passed in by the calling agency justify the transitive closure it requires before calling the inner service. When U tells A to route its reply to B, A relies on the assertion encoded in the routing instruction, $U \rightarrow B$, to deduce that for this *particular* invocation, $A \rightarrow B$.

	Goal	New Elements	New Constraints	Induced Property
R+REST	Compose services provided by multiple agencies.	RoutingProxy Component that permits clients to control relaying.	Every representation transfer must be justified by a corresponding edge in the web of trust.	*Multilateral Extensibility:* Can compose trusted invocations without requiring mutual trust.

Table 6: Summary of the R+REST style.

One significant caveat is to note that correct correspondence between RoutingProxy and *delegate()* requires path integrity. If an intermediate node fails to obey the routing instructions, or rewrites them, whether due to error or malice, our proof would be invalidated.

5.2.4 Implementation Issues

There have been two significant efforts to extend Web applications using routing rules for HTTP messages: the Protocol Extension Protocol (PEP, [80]) and the Mandatory Extension Mechanism (M-HTTP, [81]). Each of these replaced *ad hoc* proxy-selection of each hop in REST with explicit direction from the requestor. Intermediate proxies could be instructed to select one of several 'compatible' implementations of a service; if none could be found, it could also determine whether the request should continue or fail. These efforts were guided by the plug-in module interfaces available in popular web servers, such as Netscape's NSAPI and Microsoft's ISAPI. The critical improvement was that by adding mandatory error and warning semantics and richer response formats, they improved debugging regardless

of where errors occurred along the path. In the context of SOAP, this lesson resulted in adddding the MUSTUNDERSTAND header [31].

Extracting the routing logic from the underlying service implementations also enhances confidence that a ROUTINGPROXY can be trusted to enact a path. More advanced routing engines even propose to resolve constraints, load balance, and infer type conversions to select paths dynamically. This approach is familiar in workflow research [116] but is only beginning to be applied to Internet-scale software component integration. With the advent of 'web services,' interest in dynamically routing requests has inspired several proposals, such as WS-Routing [82] and 'SOAP routing' [123].

5.3 Asynchronous, Routed REST (ARREST)

Together, A+REST and R+REST provide an even more powerful abstraction for application architects: publish/subscribe integration. By combining the ability to set up a long-lived WATCH relationship with the ability to redirect those notifications to a third resource, even 3rd and 4th parties can extend deployed applications by composing an arbitrary graph of services, all synchronized by the flow of event notifications.

5.3.1 Property Specification

We would like to define an architectural style that ensures a property that goes beyond simultaneous agreement over the state of resources, to ensure *simultaneous invocation* over the output of a function of that resource. If there is an event source publishing price quotes, and a separate service that computes the present value of financing that amount on a credit card, an application architect also ought to be able to treat the output of that analysis as valid 'right now' in order to display both the full price and monthly payments side-by-side.

Stabilizing *F(X)* rather than *X* itself is only possible if *X* changes slowly enough. There are several possibilities we encounter along the way to establishing the lower bound of $1/2d$ Hz (ignoring, of course, the additional ε of time it takes to evaluate *F()* itself):

1. Fetch *X*, then evaluate *F()*: $1/4d$ Hz. Each operation takes 2*d*, but there is no guarantee that read() alone establishes consensus.
2. Fetch *X* and route to *F()*: $1/3d$ Hz. If *read()* succeeds, it takes 2*d* to send the new value directly to *F()*, and an additional *d* to return the output to the caller.
3. Poll *X*, then evaluate *F()*: $1/5d$ Hz. *Poll()* takes up to 3*d* after *X* is defined, plus 2*d* for calling *F()*.
4. Watch *X*, then evaluate *F()*: $1/3d$ Hz. *Subscribe()* takes up to *d* after *X* is defined, plus 2*d* for calling *F()*.

5. Watch X and route to $F()$: $\frac{1}{2d}$ Hz. *Subscribe()* only takes d to invoke *notify()*upon updates, and sending the new values directly to $F()$ establishes simultaneous invocation within $2d$.

Clearly, the first two are only partially satisfactory, since they do not even guarantee consensus. Furthermore, nesting additional function applications adds $2d$ each time, without taking advantage of delegation (routing).

Figure 10 illustrates how the combination of Asynchrony and Routing leads to the highest performance: triangulation. It presents five approaches to the problem of computing $F_A(G_B(X_C))$: two using *read()*, which could fail; and three using some form of *subscribe()*. In the case of REST+P, the dash-dot line shows the worst case. In order to compare characteristic update frequencies, we did not count the initial *subscribe()* requests, by assuming they were occurred beforehand.

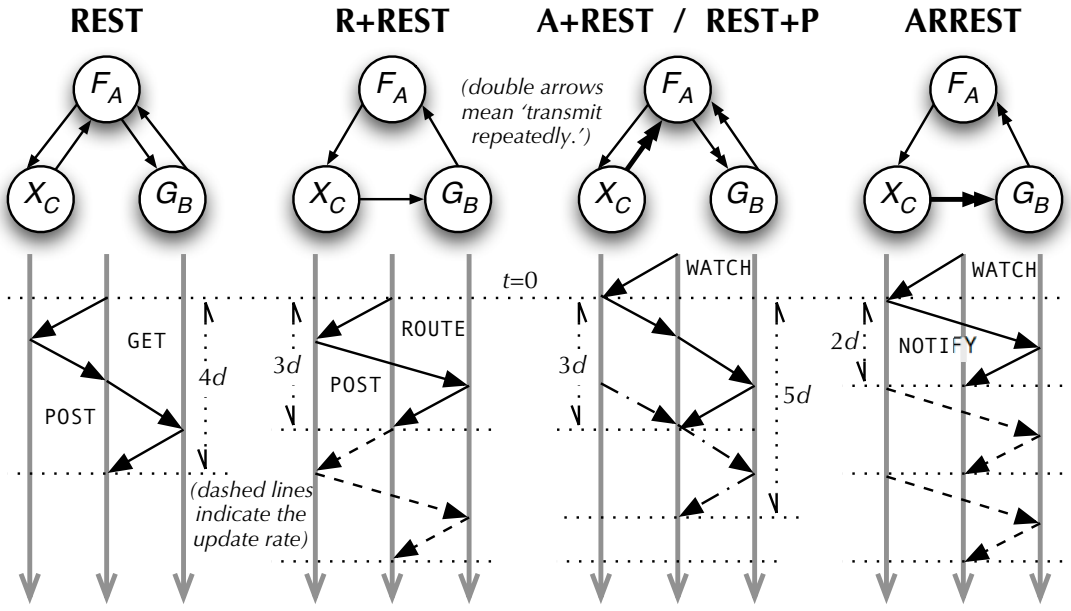

Figure 10: World-lines illustrating the total latencies for simultaneous invocation in each of our centralized styles.

Formally, though, there is no specific new facility associated with simultaneous invocation. The specification of the combination does not confer greater explanatory power. However, it does have a significant impact on the definition of the style (§5.3.2) and its implementation (§5.3.4).

5.3.2 Style Definition

> The ARREST architectural styles induces simultaneous invocation by adding a CENTRALIZEDEVENTROUTER component that can create, update, delete, and enact *subscriptions,* which are constraints that force updated representations to be relayed to all matching subscribers' ORIGINSERVERs.

The combination of both Asynchronous and Routing elements with the REST style enables centralized publish/subscribe integration. This allows followers to rely on local computations that depend on remote resources, because simultaneous invocation guarantees the architect that the appropriate event handlers will be called automatically in order to keep local results synchronized with the leader's value.

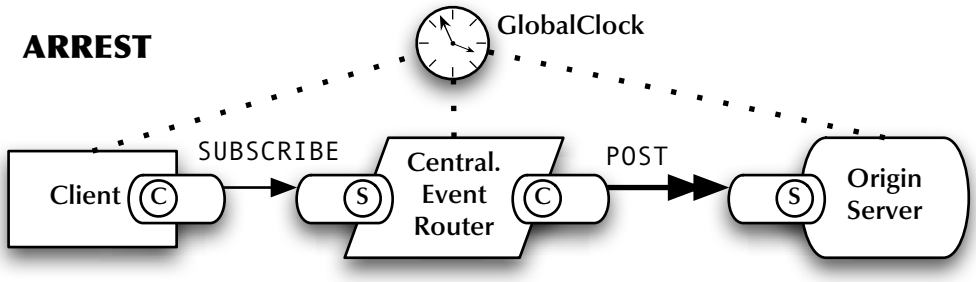

Figure 11: Illustration of the ARREST architectural style.

While there is no change in the formal specification of both asynchrony and routing, the combination of the two permits us to define a new architectural element for the sake of convenience: the CENTRALIZEDEVENTROUTER. Its most important novelty is the introduction of first-class subscriptions.

> A **subscription** is an *(S, D, L)* triple consisting of *S*, the name of source to be monitored; *D*, the name of the destination resource to be notified; and *L,* a lease indicating the interval of time when the owner of *S* intends to notify *D* of each change to *S*.

Recall that a NOTIFYINGORIGINSERVER must maintain state for the duration of a WATCH relationship, but there is no particular necessity for reflecting upon such state. In other words, since the only action that can create a relationship is the initiating client's WATCH request, there is no need for an administrative model to reason about subscriptions, nor any way for 3rd or 4th parties to create such.

On the other hand, the concept of a routing path is entirely ephemeral in ROUTINGPROXY: paths apply to a single resource request. Adding a routing path to a

WATCH request, though, can create a persistent relationship between several resources across agency boundaries. Entire application architectures can be encoded as a series of appropriately-chosen subscriptions and "drop-boxes" to store and forward event notifications.

Note that while a subscription must be owned by the same agency that owns *S*, the event source, it can be created by anyone that *S*'s owner trusts. Formally, creating a subscription does not even require the consent of *D*'s owner, because any resource must be prepared for the possibility of unwanted notifications ("spam"). This is why we speak of the ability for 1st parties (requesting clients), 2nd parties (replying servers), 3rd parties (other clients), and even of "4th parties" (administrators) to create and manipulate subscriptions.

5.3.2.1 The CentralizedEventRouter Component

A CentralizedEventRouter is a container for event sources, just as a REST OriginServer is a container for resources. However, it has an additional, stateful mode of operation. Clients can request creation of SUBSCRIPTIONS by specifying a resource to monitor, a resource to notify, and the duration of the relationship. The response is not the state of the monitored resource, as it would be with GET or WATCH, but rather the identity of a newly created resource. Unsubscription now becomes a matter of deleting this new resource. Indeed, access controls for the creation and deletion of SUBSCRIPTION resources can enact complex trust models. Finally, the creator of a subscription need not be the same agency as either the monitored or the target resources' owner: this provides for both explicit and implicit invocation of services. We call this approach "composing active proxies," because it allows architects to connect software components by redirecting the flow of messages across the network, without directly modifying any of the underlying services [124].

5.3.3 Validation

ARREST indeed induces simultaneous invocation when composing services, as summarized in Table 7. We validate this claim by arguing that, by construction, the new architectural elements and behavioral constraints of ARREST do not weaken the arguments already advanced for A+REST and R+REST in §5.1.3 and §5.2.3, respectively.

	Goal	New Elements	New Constraints	Induced Property
ARREST	Refer to the results of services that depend on centralized event sources.	CENTRALIZEDEVENT-ROUTER Component combines facilities to recast Resource/ Representation as an Event Source/ Event Notification model.	Notifications are relayed directly *through* services (proxies) to minimize latency.	*Simultaneous Invocation:* Ensure that services are invoked with the same inputs at the same time, every time.

Table 7: Summary of the ARREST style.

The key issue is whether the function being invoked can be characterized as an "event handler": non-blocking, re-entrant, and fast. To achieve simultaneous invocation of *F(X)* whenever *X* changes, we know that a NOTIFYINGORIGINSERVER and a ROUTINGPROXY can be composed to initiate *F(X)* within at most 2*d* of a change to *X*. However, *F()* itself must terminate quickly, since the total "margin of error" is only L_{min}–2*d*. Second, actual network latencies may be << *d*, so even though the update rate may be low enough, two particular invocations of *F()* may occur very close together. Hence, the requirement for non-blocking evaluation, to permit more than one outstanding invocation — to say nothing of the likelihood that *F()* may be used by many different subscriptions at the same time. Third, it should not have side-effects (re-entrancy): the same handler may get invoked again and again with the same value of *X* if the centralized server's expiration policy uses a heartbeat.

Admittedly, these are difficult requirements to test for, not least of which is the Turing-complete termination criterion. Furthermore, there may be additional levels of nesting, and every level increases L_{min} by *d*. However, for any *F()* that does satisfy these tests, CENTRALIZEDEVENTROUTER can guarantee simultaneous invocation.

5.3.4 Implementation Issues

There are two significant new implementation issues arising from combining ROUTINGPROXY and NOTIFYINGORIGINSERVER into a single CENTRALIZEDEVENTROUTER: support for the new concept of 'subscription' resources and scalability.

5.3.4.1 Reflection

Support for first-class subscriptions calls for an additional, reflective interface for introspection to create, update, and delete them. Recalling our discussion of Jabber, note that its approach violates the boundary between the application and network layers [235], both by relying on network connectivity to indicate a subscription lifetime (i.e. a TCP half-close); and by avoiding end-to-end identification by only

using ephemeral IP addresses and TCP port numbers to distinguish subscribers from each other.

In contrast, subscription resources are created and destroyed by separate network transactions; and rely on resource identifiers that have much longer lifetimes and are consistent across agencies (at least, while using DNS for host names). For example, in MOD_PUBSUB we currently attach a KN_ROUTES subdirectory to every topic (event source) to enumerate the subscriptions to it. This makes it much easier to discover what other agencies are rendezvousing at that topic. Adding recursion to reflection also provides us the ability to reason about higher-order access control. For example, denying write access to /FOO/KN_ROUTES/KN_ROUTES prevents others from discovering the list of other agents watching /FOO/.

Looking ahead, the capabilities of a WebDAV-based event router could WATCH entire collections of resources (using subscriptions that specify a DEPTH: header, say), as well as enumerating collections of subscription resources to efficiently navigate the sets of subscribers to a resource.

5.3.4.2 Scalability

The significant new opportunities CENTRALIZEDEVENTROUTER poses relate to scale: optimizing dispatching for multiple subscriptions to the same resource; and retransmission policies for offline route destinations.

The former is a classic area of research for Internet-scale event notification services: if subscriptions can be viewed as continuous, standing queries over a data stream [15], how might such queries be optimized? The IBM Gryphon project is just one example of the research interest in efficient query compilation and execution [6]. The common theme in this research area is the choice of a query language for expressing subscriptions. Given a fixed query grammar, the problem of scanning an input stream for matches can take advantage of techniques found in the extensive literature on parsing and compilation [7].

A related opportunity is currying intermediate filter output: if *F()* is a popular currency-conversion service, can its output be reused for other subscribers using the same currency pair? It seems reasonable to use existing REST caching policies for determining sufficient similarity between independent requests. On the other hand, hypertext caching may only be feasible because the range of USERAGENT behaviors is so small; arbitrary service requests may have too much context to be usefully share results with others.

Store-and-forward notification is another area ripe for engineering optimization. Just because the networks we are analyzing guarantee *d* as the maximum network latency for reliable messaging, that doesn't mean it can send a message to a recipient that's not there. R+REST already raises the possibility of routing to a destination resource-identifier that is nonexistent, or at least unavailable. While it is technically

correct to discard such notices, a practical system may want to store and retransmit notifications to allow recipient systems to survive crashes that last much longer than *d*. At that point, entirely new resource-utilization and scheduling subsystems may become necessary — this stops looking like a web server and far more like an e-mail server.

Chapter 6: Distributed Systems

In the previous chapter, we added the concept of subscriptions to REST. To complete our derivation of an event-based style from REST, we also need to add *publication*. This chapter distributes the ability to publish from a single central location to many additional locations. The effect is to distribute control over the representation of a resource across several peers.

Our goal is to add multiple publishers without sacrificing simultaneous agreement. This leads us to measure our success according to the 'ACID' properties: whether changes to a distributed variable are Atomic, Consistent, Isolated, and Durable. Since there is little any generic architectural style can do to assure consistency and durability — these depend on application-specific semantics and implementation-specific behavior, respectively — our focus shifts to ensuring serializability, since total ordering of operations consequently induces atomicity and isolation. This leads to our specific goal of eliminating the risk of 'lost updates.'

The lost update problem occurs when two clients attempt to publish a new value for the same variable at the same time: neither party would be aware of the other's work, and one party's work could be blindly overwritten. Our general approach is to add Decision functions that determine which of several peer copies is currently "in charge." However, because our model of a network for any consensus-based architectural styles is a reliable one, we can use the ersatz-distribution 'shortcut' described in §2.3.2 to update distributed resources as frequently as once every $2d$ seconds. In effect, our decision functions collapse down to centralized locks when the only type of failure to be modeled is the fail-stop process.

In this chapter, we will introduce two styles: REST+D for 2-way sharing, and AR-REST+D for N-way sharing. We add pairwise distribution to REST using locks to temporarily *delegate* control to a single client. To enable N-way sharing of control, we need a *distribution* function that can derive the common state from a vector of each peer's current state. That requires adding event-notifications to our concept of a decision function, to keep all N publishers in simultaneous agreement; as well as eliminating the need to re-read the value after locking to prevent lost updates.

6.1 Property Specification

The earliest popular distributed system was arguably the database management system. Even when applications and DBMSes ran on the same mainframe, there were still clear advantages to separating the concerns of data storage and application logic in this architectural style. However, that boundary also induced new risks due to delegation of control: what if the database itself failed?

The common measures of interchangeability are the 'ACID properties,' standing for Atomicity, Consistency, Isolation, and Durability [101]. Their general intent is to make using a remote database indistinguishable from using tightly coupled local storage. However, most practical systems have to make tradeoffs in the degree of 'ACIDity' precisely to avoid such tight coupling.

From an architectural perspective, the essential abstraction is the *transaction*: a related set of operations on a related set of resources. The archetypical mechanism for enforcing transactional integrity, in turn, is to maintain a parallel *lock* for each resource, and use any of several mutual-exclusion algorithms to arbitrate control of those locks.

We are not concerned with fairness, only safety and progress. Fair locking algorithms are necessarily stateful, so an open, Internet-scale system can only be expected to offer access to *some* client to guarantee overall progress. As for performance, we have already established the theoretical minimum: N-way shared transactions require a minimum lock lifetime of $N \cdot d$ seconds (§2.3.2); though solutions based on locks (temporary centralization) can reduce this to $2d$ seconds (§2.3.2.1).

6.1.1 Atomic

> *Atomicity*: The system under test must guarantee that transactions are atomic; the system will either perform all individual operations on the data, or will assure that no partially-completed operations leave any effects on the data. [100]

Atomicity can be induced by distributed locks. One extremely simple strategy is to lock the entire database: one lock for all resources. As long as the entire database is written back to disk before releasing the lock, then this property can be satisfied simply by performing all operations while holding the lock.

However, it is extremely inefficient to lock an entire server. To minimize contention and maximize concurrency, architects often desire finer-grained locking schemes. However, that could cause an architect's intended transaction to require atomic operations on several resources at the same time. Not only must a component ensure that it can eventually acquire all of the locks it needs, we shall see below that the Isolation and Consistency properties also require a mechanism to release all those locks simultaneously. This problem is known as a *compound transaction*, often implemented using an external *transaction monitor* [101].

To enable compound transactions, it is also critical for the architect to totally-order every lock, in order to ensure every participant in a transaction prioritizes lock acquisition identically. This approach is also known as token-coloring, based on the literature of solutions to the "drinking philosophers" problem [45].

For all these reasons, we do not tackle this problem directly in this chapter: we will presume that architects can represent compound resources as yet another (singleton) resource. A brief example would be the choice to represent a customer's billing address as a single SHIPTOLOCATION resource rather than independent CITY, STATE, ZIP, ... resources.

6.1.2 Consistent

> *Consistency*: Consistency is the property of the application that requires any execution of a transaction to take the database from one consistent state to another. [100]

Simply put, there is nothing a domain-independent architectural style *per se* can provide to enforce consistency — even with complete serialization.

The archetypal example of consistency is the conservation of funds in a banking application. Before and after a money transfer between two accounts, the total balance of the bank must not change — it must not be possible to observe an "in-between" state where funds have been debited from the source without being credited to the destination yet.

However, a naïve implementation of compound transactions would not enforce such conservation. The application could still write an incorrect balance into the destination account and it would still be a valid transaction because the in-between state was not visible.

Consistency should properly be considered a domain-specific property induced by the actual component logic — not the architectural style, which must necessarily be generic.

6.1.3 Isolated

> *Isolation*: Operations of concurrent transactions must yield results which are indistinguishable from the results which would be obtained by forcing each transaction to be serially executed to completion in some order... This property is commonly called serializability. [100]

This property specifies the consequences of concurrent accesses: they must not appear to be concurrent. Ensuring that all updates to shared resources are isolated is tantamount to simultaneous agreement. Therefore, one solution is not merely to ensure that updates *appear* to be serialized, but that updates are *actually* serialized.

A mechanism that enforces this requirement is simply to delay pending updates long enough to ensure that the update rate does not exceed the theoretical maximum. Then, the mechanisms we have already introduced for establishing simultaneous agreement have time to execute.

It is possible to increase concurrency beyond this limit if separate transactions depend on non-overlapping subsets of a database — the same challenge of determining an appropriate granularity we encountered in §6.1.1. However, it is not clear that a completely domain-independent architectural style can achieve that degree of isolation automatically. Dependencies between resources are necessarily determined by domain-specific application semantics.

6.1.4 Durable

> *Durability*: A transaction is considered committed when the transaction manager component of the system has written the commit record(s) associated with the transaction to a durable medium. [100]

Durability is largely an implementation issue, because it is a matter of degree. Durability measures a tradeoff between failure and performance that software architects (and end-users, at run-time) must make consciously:

> No system provides complete durability, i.e., durability under all possible types of failures... A durable medium can fail; this is usually protected against by replication on a second durable medium (e.g., mirroring) or logging to another durable medium. [100]

Indeed, the latter point is the primary architectural decision that affects the property of durability: whether or not to trust the network. In theory, a fully reliable network ensures that merely broadcasting a commit-event notification to a set of replica servers will eventually lead to a durable write on each server after $d+\varepsilon$ (where ε is the storage delay). In practice, a fallible network calls for a solution such as a two-phase commit protocol between all the replicas, at the expense of increased message traffic and total round-trips.

Another way of expressing this architectural decision is by choosing the level of abstraction of a "durable medium." The first case resembles drawing an architecture diagram with a RAID device in place of a single disk: architecturally this merely replaces a faulty component with a less-faulty one (albeit with differing latency, bandwidth, and cost). The opposite of such black-box substitution is a drawing with several parallel components that jointly share control over a record using an explicit decision function. Ultimately, this is a recursive argument: distributed control of a variable is automatically more durable by virtue of eliminating a single-point-of-failure.

The only other architectural feature that influences the property of durability is simply the number of copies. Assuming that critical faults in durable storage are randomly distributed and independent, distributing control across additional replicas will increase durability of the entire system.

6.2 REST with Delegation Decisions (REST+D)

The earliest efforts to extend the Web to support authoring clients immediately ran afoul of the "lost update" problem. To the degree that the server's copy of a file was the sole authority for a resource's representation, two editors using local, cached copies could overwrite each other's work. This limitation of HTTP is reflected in REST as well.

In this subsection, we derive a new style that extends REST to induce ACID simultaneous agreement for pairwise distributed resources (that is, where authority for a resource's representation is shared between a client and the server). More specifically, our goal is a style that enforces Atomicity and Isolation automatically, while still enabling architects to maintain (application-specific) Consistency and Durability properties.

6.2.1 Style Definition

> The REST+D architectural styles induces simultaneous agreement for a 2-way *distributed* resource using a new component, MUTEXLOCK, that enforces the constraint that only one component may modify the representation of a resource at a time.

Precisely because REST explains much of the design of HTTP, it is instructive to consider why WebDAV [60] needed to modify the design of HTTP/1.1 so extensively in order to support distributed, collaborative authoring.

In HTTP/1.1, there is no mechanism for a client to explicitly modify a resource on the server. Even with rudimentary PUT support, updates can be lost, since there is no notification mechanism to keep a local copy synchronized with a remote resource while it is being edited. Furthermore, another practical challenge was its support for the legacy HTTP/0.9 behavior of using TCP connection termination to indicate the end of a message. This meant that an aborted upload could replace the current representation, violating the Atomicity requirement.

In addition to mandating atomic behavior for resource creation and modification, the WebDAV standard also created a new LOCK method for HTTP/1.1. For each resource, a WebDAV server must also maintain a stateful lock record to track which client has exclusive write access at the moment. Therefore, locks explicitly delegate control over a resource from the server to an external client. Of course, since there is no notification mechanism to maintain simultaneous agreement for other, read-only, clients, WebDAV can only fully support 2-way shared resources (where one of the controllers is always the server).

Another fundamental shift in WebDAV is the creation of first-class COLLECTION resources that aggregate multiple REST resources. In ordinary Web usage, the slash

('/') is just another character in a URL, one that only coincidentally might correspond to a server's internal abstraction of files-and-directories. By contrast, COLLECTIONs allow users to reason about hierarchical containment of resources *on the same server* (in conjunction with the DEPTH: header). Note that this is the only form of aggregation WebDAV protocol supports directly; there is no way to atomically update multiple resources that do not share a common pathname prefix.[15]

Rather than continuing to describe extensions to the REST architectural style that would be detailed and powerful enough to explain all of the new concepts found in WebDAV, we opt to define the only additional element we need to support ACID transactions: MUTEXLOCK. As shown in Figure 12, MUTEXLOCK is a proxy that wraps around an ORIGINSERVER to restrict access to one CLIENT at a time.

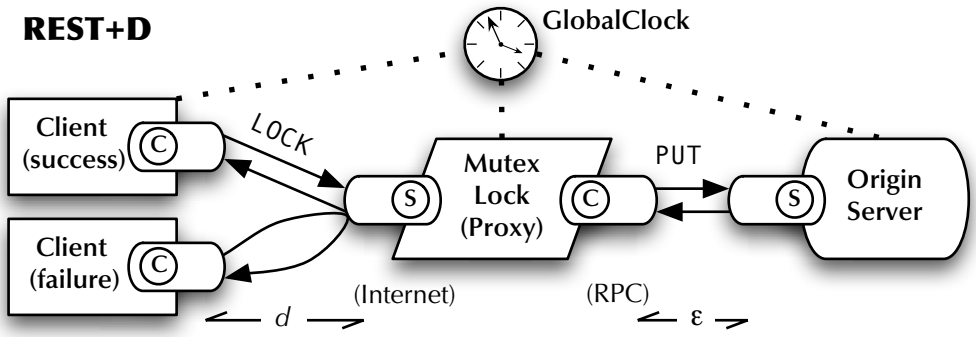

Figure 12: Illustration of the REST+D architectural style.

6.2.1.1 The MUTEXLOCK Component

The MUTEXLOCK Component ensures mutually exclusive access to an ORIGINSERVER. It contains an atomic test-and-set register identifying the only client whose request messages it will forward on to the ORIGINSERVER until instructed otherwise, or until the lease expires. All other requests are simply discarded until the register is reset.

This solution presumes that an ORIGINSERVER has at least some primitive PUT capability with an atomic commit (as a safeguard against aborted transmissions). MUTEXLOCK also requires a finite maximum lease duration for locks, in order to prevent a crashed client from bringing the entire system to a halt.

[15] Ongoing work on the Delta-V versioning protocol extensions to WebDAV [50] do at least raise the possibility of "external collection members," but still with the caveat that atomic updates can no longer be guaranteed across server boundaries.

6.2.2 Validation

As [100] itself notes, "No finite series of tests can prove that the ACID properties are fully supported." Our argument for the validity of REST+D, therefore, rests upon establishing that MUTEXLOCK fulfills the requirement of a 2-way transaction manager, which we already established would uphold the relevant ACID properties. As summarized in Table 8, our new style for REST with Delegation (REST+D) prevents lost updates by using a lock for each resource, and enforces complete Isolation by re-reading resource state once the lock has been acquired.

	Goal	New Elements	New Constraints	Induced Property
REST+D	Refer to a pairwise distributed read/write resource reliably.	MUTEXLOCK Component ensures only one client at a time has write access to the origin server.	Lock must be acquired before attempting write; then current state of the resource must be re-read before writing.	*ACID (Pairwise) Simultaneous Agreement:* Clients can modify centralized resources within $3d$ — but only in the absence of contention.

Table 8: Summary of the REST+D style.

Our obligation to provide an implementation for a 2-way decision function is satisfied by MUTEXLOCK. Essentially, the decision function takes the value of the central lock arbiter as input to a mutual exclusion protocol to determine which component is currently "in charge," and we use that component's value as the correct current value.

In the balance of this subsection, we will argue for the safety and progress properties of the REST+D style.

SAFETY. Acquisition of a lock is our new precondition for issuing a PUT request in the REST+D style. Assuming that MUTEXLOCK maintains a separate lock record for every resource on the ORIGINSERVER (which must be arranged in a tree — no "symbolic links" that could form cycles), then there are two cases. First, if the lock record is clear, a single test-and-set operation suffices to assign the lock to the requesting client, and the encapsulated PUT request is pipelined onward to the ORIGINSERVER. If it is not clear, then we can know it must eventually become clear, no later than L_{max}, the maximum lock (lease) lifetime.

However, note that in the description above we still have not proven safety according to the ACID properties. The problem is that "pipelining" together a lock request and a write request in one shot does not verify the precondition that the write is replacing the immediately preceding state — it could lead to a lost update. Isolation requires the client to *re*-read the state of the resource after acquiring a lock

(unless a write is completely independent of the past state, an application-specific exemption for architects to consider optimizing for).

Since the MUTEXLOCK discards *any* request from any client other than the lock-holder, including GETs, it functions as an exclusive-read, exclusive-write lock (EREW, in Parallel Random Access Memory (PRAM) parlance, [111]). Thus, the maximum update rate is determined by the minimum lock lifetime, since the update can't "take effect" until other clients can read it. This reduces the potential performance of REST+D by requiring at least $3d$ between updates (as compared to the theoretical best case of $2d$).

Consider how long it takes to arbitrate between two writers that both wish to update a resource at the same moment (and do not even care about lost-updates): the first "winner" sends a single pipelined LOCK/PUT/UNLOCK request, which takes only d. In the meantime, the second writer's request is blocked; the soonest it could be sure to proceed is if the server immediately notified it of the LOCK expiration, which would take d. Then, it would take a third network trip for the new value to actually arrive at the server, adding up to a total delay of $3d$ for a 2-way shared update.

PROGRESS. If the corresponding lock record is not clear, the current client's request will fail. However, eventually *some* other client will be able to acquire the lock. This segues into the second portion of our argument: MUTEXLOCK may not guarantee fairness, but it does guarantee progress. One significant reason is that most fair solutions to the distributed mutual exclusion problem require maintaining additional state information at the server to prioritize among an unbounded range of clients.

Thus, to argue for the validity of REST+D, we must only argue that the maximum delay for *some* client to acquire a lock is finite. For any given client process, we can indeed construct a counterexample that interleaves steps taken by other clients that leads to an infinite delay in lock acquisition. All the same, the maximum duration that a lock can remain clear when there are pending clients is clearly not infinite — the very next client that requests a lock must succeed.

6.2.3 Implementation Issues

Any practical implementation of the REST+D style faces two separate implementation challenges: minimizing the latency "between" the MUTEXLOCK proxy and the ORIGINSERVER; and ensuring durability over a fallible network.

LATENCY. Note that in Figure 12, there is a note below each set of network connections that indicates that the CLIENTs are connected to the MUTEXLOCK over the Internet, with a maximum delay of d, but the MUTEXLOCK itself is very tightly coupled to the ORIGINSERVER, minimizing the additional delay it causes to at most 2ε. For example, locking is often built directly into a WebDAV repository, making $\varepsilon << d$.

DURABILITY. Resource updates must be committed to stable storage before generating any successful response message. This requires accounting for the latency due

to interactions with stable storage devices (disks, tapes, etc.) Under certain conditions, this could be a significant multiple of *d*.

Furthermore, since storage devices can fail, too, merely presuming that the network reliably delivers write request messages is not sufficient either. There must to be some way to signal the client that a commit has occurred.

To be sure, WebDAV does not make any normative claims about durability before replying successfully. One of the only application-layer transfer protocol standards that does, by contrast, is the Simple Mail Transfer Protocol (SMTP). It obliges the receiving mail server to commit a message to disk before any Mail Transfer Agent (MTA) can reply `"200 OK"`. This is essential for a store-and-forward email system, because only a successful reply can relieve the sender of its duty to attempt retransmission (although even that duty typically lapses after a few days; SMTP can still silently lose email messages).

6.3 Asynchronous, Routed REST with Distributed Decisions (ARREST+D)

A single, centralized server poses a risk: if it fails, the entire application architecture fails. The alternative is to distribute that task across several servers. To the degree that system failures are independent, greater distribution implies greater reliability and availability. Majority voting is just one simple example of a shared decision function that can recover from the failure of up to $\lfloor N/2 \rfloor$ processes.

Another straightforward solution is *replication* from a master server to a set of slaves, using a leader re-election process to transfer control to a backup server in case the master fails. This requires an event notification facility to ensure that slaves are always "in sync."

Replication can even achieve higher performance than delegation did. It can allow updates as rapidly as every 2*d* seconds — because enforcing Isolation doesn't require an additional round-trip to read the current value. Piggybacking the lock grant on an event notification messages establishes simultaneous agreement as soon as the client gains the lock.

Recall, though, that replication is still an ersatz form of distribution, since it actually requires designating a central point of control at all times. In practice, messages lost in transit during a server failure cold still cause that entire system to halt.

We have already specified an architectural style that adds event notification to REST; to derive ARREST+D, we will show how to induce ACID simultaneous agreement for *N*-way distributed resources (that is, where authority for a resource's representation is shared across a set of *N* peers).

6.3.1 Style Definition

> The ARREST+D architectural styles induces simultaneous agreement for an *N*-way *distributed* resource using a new component, FAIRMUTEXLOCK, that enforces the constraint that only one component may modify the representation of a resource at a time *and* that every component could modify it *eventually*.

The decision function we intend to implement with ARREST+D is a *fair* mutual exclusion protocol. Based on inputs from each process indicating when it declares interest in locking the shared resource, the decision must cycle fairly through all the contenders. This way, we can preserve the best-case results of REST+D when there is no contention, while still bounding the worst-case for any particular client by bounding the number of times it can be bypassed in lock acquisition.

A simple implementation of this process is to couple a MUTEXLOCK with a FIFO queue, rather than summarily rejecting lock requests while the lock is already outstanding. We will show how to implement the combination of these two facilities, called FAIRMUTEXLOCK, by maintaining a copy of the queue at each publisher.

Figure 13 depicts how this component fits into the ARREST+D architectural style. In the abstract, it acts as a crossbar switch to ensure that every read and write access to a resource is mirrored between three servers. Unlike previous styles, each agent must actually maintain an independent resource, since the 'distributed' resource is actually calculated by applying a decision function over all three replicas.

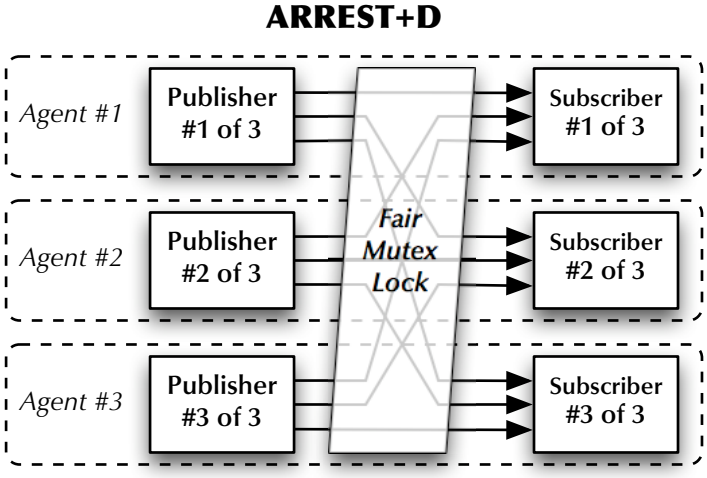

Figure 13: Illustrated example of a 3-way shared resource in the ARREST+D architectural style.

6.3.1.1 The FAIRMUTEXLOCK Component

We must transform the MUTEXLOCK component so that it can manage contention between all N potential publishers. Continuing to use an unmodified MUTEXLOCK risks starvation for some publishers; there is no mechanism to rule out traces that bypass some contenders infinitely often.[16] Conversely, our abstract specification of the behavior of a FAIRMUTEXLOCK component is that the bypass number be $< N$. This is a strong fairness requirement; it implies that no client may re-acquire the lock if any other client is still waiting.

A centralized implementation of FAIRMUTEXLOCK would be to couple MUTEXLOCK with a FIFO QUEUE. Such a QUEUE would be a proxy server that simply buffers its input requests and delivers them one at a time to the downstream server (in this case, the MUTEXLOCK proxy server). Of course, this only makes the Queue another bottleneck, which we shall eliminate below.

The second major change required is to replace the ORIGINSERVER with a CENTRALIZEDEVENTROUTER. By requiring all N publishers to subscribe to the shared resource at the CENTRALIZEDEVENTROUTER, each of the local proxy variables remain in simultaneous agreement (and hence, the decision function remains stable, too). In effect, this scenario adds notifications to an ordinary WebDAV server.

Returning to QUEUE, however, we still must replace it with a genuinely distributed mutual-exclusion protocol. A classic example of a distributed decision function is Lamport's Bakery algorithm for arbitrating access to a critical section [136]. The original is presented for a process P_i in Program 17: contending processes claim a monotonically-increasing ticket number (the "doorway"), and then wait their "turn" to enter the critical section.

$choosing_i$:= *true* ;
$ticket_i$:= max($tickets_{k \neq i}$) + 1;
$choosing_i$:= *false* ;
busy-wait-until ($\forall\ k : k \neq i$; $choosing_k$ = *false*)
    busy-wait-until ($\forall\ k : k \neq i$; (($ticket_k$, k) > ($ticket_i$, i)
                                    $\vee$ ($ticket_k = \varnothing$))
        critical section ;
        $ticket_i := \varnothing$;

Program 17: Typical presentation of Lamport's Bakery algorithm.

[16] This formulation of fairness, in terms of the number of times a process may be passed over, is apparently due to [39].

The Bakery algorithm has since inspired many variants [11]. Perhaps its most compelling property is that it can implement distributed mutual exclusion *without* relying on any lower-level atomic operation (as many once thought necessary [138]). This suggests that it can, indeed, be transformed from a shared-memory algorithm to a synchronous network protocol. Presuming a finite d for message latency, all we need to do is insert a pause after the doorway, to ensure that the writes have time to propagate before the busy-waiting loops. Furthermore, since we can rely on a GLOBALCLOCK component, we can also use physical timestamps to simplify it as in Program 18:

$ticket_i$:= NOW ;
wait-until d ;
busy-wait-until ($\forall\ k : k \neq i$; ($(ticket_k, k) > (ticket_i, i)$ $\vee$ $(ticket_k = \varnothing)$)
    critical section ;
    $ticket_i := \varnothing$;

Program 18: Simplified Bakery algorithm using a clock and a network.

6.3.2 Validation

As summarized in Table 9, our new style for Asynchronous, Routed REST with Delegation (ARREST+D) prevents lost updates by sharing a lock for each resource fairly. We aim to validate that ARREST+D automatically enforces complete Isolation by ensuring that all local copies of the leader's variable are in simultaneous agreement when leadership passes to the new lock-holder.

	Goal	New Elements	New Constraints	Induced Property
ARREST+D	Refer to an N-way distributed read/write resource reliably.	FAIRMUTEXLOCK Component to arbitrate shared locks with bounded bypass.	For a peer-to-peer solution, all N must cross-subscribe to each other's centralized ticket variables.	*Atomic, Isolated, Durable*: Ensure updates apply to all N resources reliably within $2d$ and at most $N \cdot d$.

Table 9: Summary of the ARREST+D style.

Our obligation to provide an implementation for an N-way *df()* is satisfied by FAIRMUTEXLOCK. Essentially, the decision function takes the lock-requests of each component as input to a mutual exclusion protocol that returns which component is currently "in charge"; and we then use that component's value as the correct current value.

The remaining question concerns the performance of this architectural style: what is the minimum interval it permits between updates to a distributed variable? Surprisingly, the bound is not as high as originally discussed in §2.3.2. While any *particular* client may get stuck waiting for all N others to write first, *some* client will always be able to acquire the lock.

The reason our decision function produces a stable output so quickly is that the inputs are not allowed to change in ways that would affect its output. Newly-minted tickets have monotonically increasing timestamps, and also cannot be canceled until that lock is eventually granted. Therefore, the ordering of the queue is stable after only d seconds have elapsed.

Of course, another way of looking at this is that we have neatly sidestepped the problem of solving the peer-to-peer mutual exclusion problem by positing an external solution: the global clock itself. A completely clock-less solution would face $N \cdot d$ delays for the decision function to stabilize, as commonly found in hardware communication buses [212].

The second reason that ARREST+D can achieve a lower minimum bound between updates of $2d$, rather than REST+D's $3d$, is that event notification eliminates the extra hop of requesting that the current value of the variable be transmitted to the new lock-holder. At least when using the distributed BAKERY solution, cross-subscription[17] between all N peers allows update notifications to go directly to the new lock-holder without being relayed via a central server.

6.3.3 Implementation Issues

Whereas an implementation of REST+D requires an actual MUTEXLOCK to arbitrate delegation decisions, we propose an end-to-end implementation of FAIRMUTEXLOCK that will not require modifying CENTRALIZEDEVENTROUTERs at all. Program 19 shows how to implement the queue as a set of local variables at each publisher, using acknowledgment messages to avoid characterizing d. This approach actually dates back to an even earlier work by Lamport, the mutual exclusion example given in the classic paper *Time, Clocks, and the Ordering of Events in Distributed Systems* [134].

The program presumes that each publisher has already subscribed to each of the topics it has a *Handler* for. Each publisher maintains a vector of acknowledgments and a queue of pending lock requests. When any component needs to modify the shared variable, it notifies the group of its interest; waits for acknowledgement of its "ticket" (in Bakery-speak); and then waits until it has the oldest outstanding ticket.

The function we did not specify, though, is *updatedValue()*. The key to eliminating lost updates is that the "user" cannot specify the new value immediately upon re-

[17] This presumes that all N peers know of each other in advance; dynamically adding or subtracting members from that set complicates matters further.

questing a lock. Instead, the new value must be re-calculated in case it depends on the current value of the variable at the time the lock is actually granted, which might be considerably later.

```
LockRequest ()
    Q[ i ] := NOW ;
    Publish("/Foo/LockRequest", (i, Q[ i ]) );

LockRequestHandler (sender, timestamp)
    Q[ sender ] := timestamp ;
    Publish("/Foo/LockRequestAck", (sender, NOW) );

LockRequestAckHandler (sender, timestamp)
    Acks[ sender ] := timestamp ;
    CheckLock();

LockReleaseHandler (sender)
    Q[ sender ] := ∅ ;
    CheckLock();

CheckLock ()
    If (∀ k : k ≠ i ; (Acks[ k ] > Q[ i ]) ∧
                    ((Q[ k ], k) > (Q[ i ] , i) ∨ (Q[ k ] = ∅) )
        Publish("/Foo", updatedValue() );
        Publish("/Foo/LockRelease", i );
```

Program 19: Pseudocode for implementing a FairMutexLock peer.

Chapter 7: Consensus-Free Systems

At this point, we reach the limits of consensus-based styles operating on partially-synchronous networks. This is the boundary separating centralized and distributed systems — where a variable has only one value at a time — and the fresh challenges raised by consensus-freedom.

In fact, our analysis is literally based on the concept of a boundary: there is a 'now horizon' that separates the set of components that can reason about the value of a given variable 'right now' from those that can only refer to its value 'back then.' Beyond the now horizon, representation transfers that once could be treated as ACID cannot do better than our so-called 'BASE' properties.

This chapter discusses the generic challenges of decentralization; we will return to deriving specific architectural styles that incorporate these findings in the subsequent two chapters.

7.1 The 'Now Horizon'

Simultaneity is a useful shortcut for programming distributed systems, but it can only go so far. The question is, How far?

It is straightforward to determine whether a local resource can be in simultaneous agreement with a remote resource: the two components must trust each other and use a network whose round-trip time is faster than the shortest interval between updates.

Another perspective is to compute the dual of this rule: every resource defines the subset of other components that can possibly establish simultaneous agreement. We call this frontier of possibility the 'now horizon,' because only inside of it can one speak of that resource's representation 'right now.'

Beyond that boundary, its current value can only be approximated using a variety of estimation methods. That estimate, in turn, constitutes an independent, locally-owned resource with its *own* characteristic 'now horizon' for further processing.

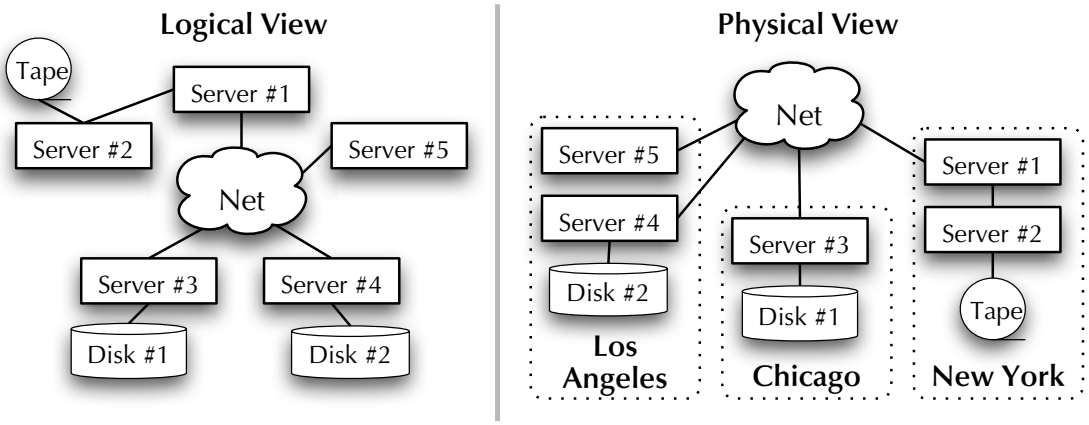

Figure 14: Logical and physical views of a transcontinental network.

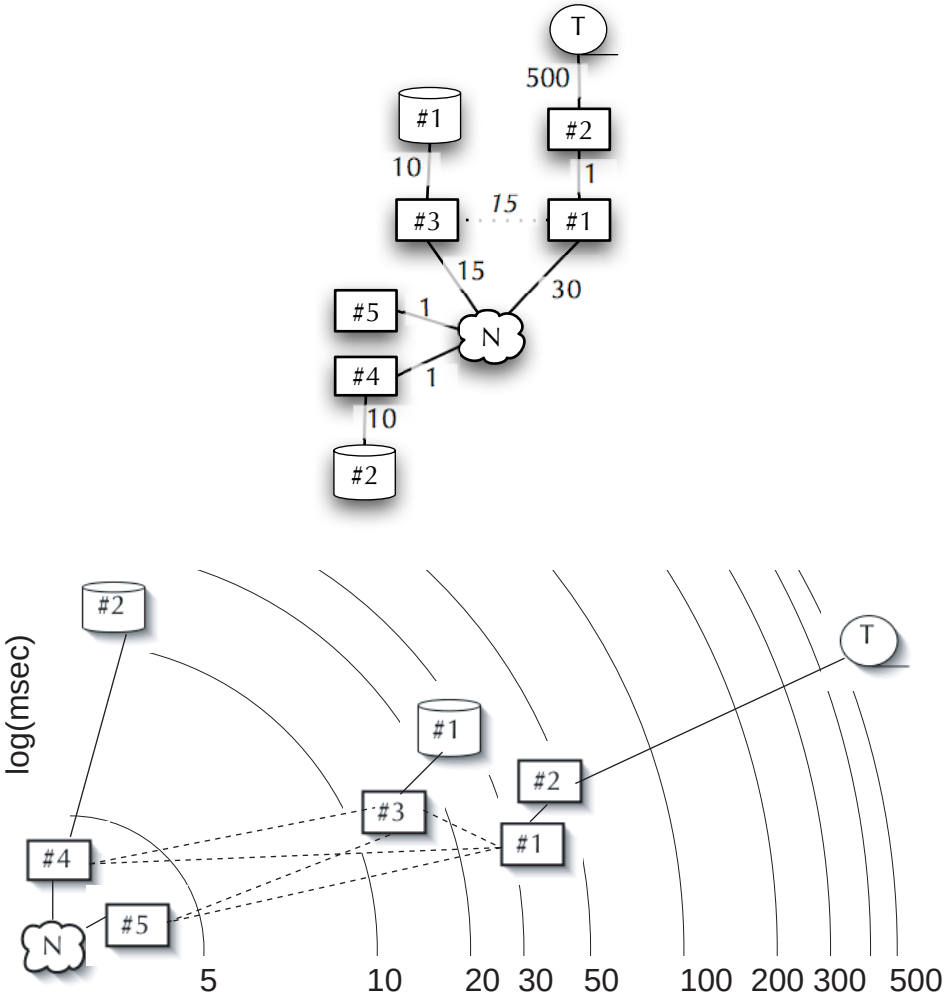

Figure 15: Latency maps of the same transcontinental network.

It is possible to map out the components of an application according to the latency between the underlying hardware in order to draw visual conclusions about the feasibility of simultaneous agreement. The key step is transforming ordinary space, where a disk might sit a few inches from a processor, into a gauge of time, where the disk may be tens of milliseconds away. For the same reason, another processor in a city thousands of miles away could still be "closer" in time than a local disk.[18]

Consider the example of an application running on data centers in Los Angeles, Chicago, and New York. Figure 14 depicts the logical layout of the network — four centrally connected servers with two disks and a dedicated tape backup server — as well as the physical assignment of devices to each city. Neither of these conventional views informs an architect of the relative communication latency between these components. Figure 15 presents two different views of that network: as a component graph whose edges are weighted by the maximum transit time, in milliseconds; and as a "latency map" depicting the total distance of each component from the network.

Note that visualizing latency calls for a logarithmic scale, since we wish to model everything from day-long email delays to microsecond-scale operating systems thread-switching delays. Interprocess communication (IPC) across a LAN is merely another kind of delay, indistinguishable from a few extra kilometers of wiring.

Map in hand, note that a circle around an arbiter whose radius is the minimum lease time of the variable traces out a boundary between components that can achieve simultaneous agreement and those that cannot. For simplicity, we are also assuming that all components trust each other in this example; otherwise, there would be gaps in the now horizon corresponding to regions controlled by untrusted agencies.

The value of this approach is that it can help architects diagnose whether or not consensus-based styles are feasible for a given application. By identifying the resources to be shared and their characteristic update frequencies, we can analyze various hardware configurations or conversely, determine the maximum update frequencies centralized or distributed solutions could bear.

Returning to our example, imagine that the application is controlling a natural gas pipeline network. A pressure sensor in Los Angeles that is being read 100 times per second cannot be shared outside of LA. To confirm this, draw a 10 millisecond (100 Hz) circle centered on LA, and note which components fall outside that boundary.

---

[18] Jim Gray uses a vivid illustration of memory hierarchies: "To put this in human terms, our clocks run in minutes. If I ask you a question, you say: "Just a minute," you compute and you tell me the answer. If you do not know, you ask someone in this room (cache) and it is two minutes. If you have to go to cache it is like going somewhere on your campus. If you have to go to main memory, it is an hour's drive there and back. If you have to go to disk, it is 2 years (like going to Pluto!), and if you have to go to [tape] it is like going to Andromeda..." [99]

This makes it immediately evident that feedback for that valve cannot be controlled in New York. A centralized solution can't establish simultaneous agreement at that frequency across the continent, even at the speed of light.

Another example reveals that even if the entire system required distributed control based on the concurrence of operators in all three cities (to ensure a precise balance between input and output gas flows), tape backup records could still be falsified in case of an accident. Since 3-way shared control of a resource with 30 msec maximum latency could occur within as little as 90 msec, it is clear that a corresponding 10 Hz resource cannot be backed up to tape with certainty before the state changes again.

In fact, the now horizon concept can be used to work backwards and ask, If regulators require every action in the system to be logged in real-time, how fast can the whole system react? The authoritative representation of a resource at any given moment now has to be centered on the tape drive itself. Since the maximum latency to any point in the system is 542 msec[19], and any actual control decisions would be taken by a server (not the backup server itself), the maximum feedback frequency is roughly 1Hz. With this analysis, the architect can go back to the user and check whether such a design would assure adequate safety.

7.2 BASE Representation Transfers

If 1 Hz is not safe enough — the desired system can't be inscribed within a single now horizon — then the architect must consider different architectural styles that cope *without* consensus.

A familiar manifestation of this problem is when an already-deployed system starts bogging down because atomic transactions stop working at Internet-scale. Nonetheless, transactions have become a popular mechanism for coordinating distributed systems precisely because they mask the effects of distribution. Indeed, the classical measures of a client/server interaction are the ACID properties: Atomicity, Consistency, Isolation, and Durability. Together, they aim to maintain the

[19] This figure was computed by using the all-points-shortest-paths algorithm for the weighted graph in Figure 15. Note that latencies between nodes cannot be measured by the distance between them on that graph. Instead, latency must be measured with respect to some central arbiter.

It is not possible, in general, to embed a weighted network latency graph into two- or three-dimensional space so that all pairwise distances are consistent. That is why we added the dotted lines, to designate direct network transit paths from New York to Chicago. This difficulty remains even when calculating distance using a different norm (e.g. Manhattan 'city block' distance rather than Euclidean geometry).

illusion of total serialization in the face of the twin challenges of latency and agency. That makes the ACID transaction model antithetical to decentralization, which requires permitting independent agencies to maintain multiple, simultaneously valid world models.

Our key insight is that decentralized systems can actively manage the risk of disagreement by seeking approximate, rather than simultaneous, agreement. Recalling our definition of representations from §2.3.6, we observed that in any consensus-based style one must discard representations made by untrusted agencies, or that have expired. Rather than treat representation transfers as rigid, ACID transactions, one can extract approximate information even from representations that arrive late or from external agencies.

Uncertainty arises from three physical limits to *precision*: noise, power, and relativity; and one social limit to *accuracy*: sovereignty of independent agencies.

Figure 16: An illustration of precision and accuracy using a dartboard.

We can still extract *some* meaning by casting other agencies' past representations as merely Best-effort, Approximate, Self-centered, and Efficient observations of a time-variate phenomenon, as summarized in Table 10.[20]

	Limit	Property	Constraint	Mechanism
Physical	Noise	Best-effort	Minimize latency	Retransmission
	Power	Efficient	Minimize buffering	Summarization
	Relativity	Approximate	Minimize error	Prediction
Social	Sovereignty	Self-centered	Ignore spam	Trust Management

Table 10: Summary of BASE constraints and mechanisms.

[20] Note that our use of the acronym BASE differs from two other groups': Basically-Available, Soft-state, Eventually-consistent in [36]; and Byzantine Fault Tolerance with Abstract Specification Encapsulation from [193].

7.2.1 Best-Effort

> **Best-effort** representation transfers may be lost, delayed, or reordered.

All communication channels are limited by noise, and hence confront a tradeoff between error rates and transmission rates [201]. Describing a network as "best-effort" is another way of stating $d = \infty$ — leaving us no way to distinguish between lost and delayed messages. To make progress, we must minimize $\bar{d}$, the *average* latency encountered.

One way to take a long-tailed random distribution and reduce its variance (and hence, its mean) is to repeat the process: if we transmit more copies, the *earliest* time that one of the messages will be received will keep decreasing. However, sending multiple messages, whether by intention or by chance, both reduces total channel capacity (bandwidth) and requires additional memory at the receiving end to ensure that subsequent copies are discarded, rather than treated as new messages.

The fundamental strength of IP internetworking is its "best-effort" store-and-forward message delivery model. It accepts that there is some probability of total message loss, but pushes the burden of retransmission and reordering, if necessary, out to the edges of the network.

The same identification mechanism for duplicate-suppression is also necessary for ordering messages reliably. Since a best-effort network is not isochronous, message delivery times do not reflect message transmission times (i.e. messages can be re-ordered). We can either presume the use of synchronized clocks to compare absolute timestamps, or reply on sequence numbers with a "sliding window" of unacknowledged identifiers [224].

7.2.2 Approximate

> **Approximate** representation transfers may expire.

All communication channels are limited by relativity, and hence cannot transmit information faster than the speed of light in that medium [66]. The actual velocity of information can be several orders of magnitude slower (e.g. email). High latency can thus delay a message past its duration of validity.

Nevertheless, even past observations can help minimize the error between the current value of the variable it represents and its local estimate. A prediction function is measured by the amount of error that can be tolerated while still constituting "approximate agreement." The ideal predictor, by our specification, achieves a *P%* probability of agreement, based solely on regression against past observations.

7.2.3 Self-Centered

> **Self-centered** representation transfers must require the recipient to trust the sender.

Communication channels assume the receiver trusts the messages transmitted through it. This bypasses the sovereign independence of the recipient to establish its own trust relationships with the various possible senders. To prevent abuses of trust, we need mechanisms that ensure messages transmitted on a channel were legitimately initiated by the purported sender.

Self-centered trust management can filter out messages from trusted correspondents and reject messages from untrusted or partially-trusted correspondents. Formally, self-centeredness is specified in the same manner as multilateral extensibility: every inbound communication must be justified by a corresponding edge (or path) in the receiving agency's web of trust.

7.2.4 Efficient

> **Efficient** representation transfers must not require buffering.

It is impossible to sustain an information transfer rate in excess of a channel's capacity. Unchecked, guaranteed message delivery can inexorably increase latency. To ensure this does not occur — ensuring that buffers, even if necessary, remain finite — information buffered for transmission may need to be summarized, updated, or even dropped while still queued.

Another way of specifying this limitation is that, if a channel were abstracted as a 2-way shared resource, the maximum update frequency of the channel must exceed that of any of the resources being shared across it. For example, any effort to transmit ten keystrokes per second on a one-second latency link would have to resample the 10 Hz keystrokes into "ten-character strings" sent at 1 Hz. Such coalescing of data while a round-trip time elapses is similar to the Nagle algorithm for delaying short TCP segments [164].

There is more to efficiency than minimizing the overhead of underlying communication protocols. The very format of a representation could be compressed, such as if the keystrokes actually spelled out English words. Furthermore, the semantics of a representation could be relied upon to cancel out compensating transactions (e.g. pressing a key followed by the delete key might result in zero messages being sent — for a word processor; by contrast, a videogame may require both keystrokes). This property could also interact with approximation to summarize data in ways that assist prediction by recipients, such as minimizing aliasing effects by smoothing

audio or video streams (rather than blindly resampling them at randomized intervals).

7.3 Risk Management

The broader impact of replacing ACID transactions with BASE representations is a shift from processing *facts* to *opinions*. This is essential when integrating software services run by multiple agencies across the Internet. There must be some way to incorporate the opinions of trusted correspondents into local, independent estimates in a consensus-free system; otherwise there would be no connection at all between a set of completely isolated applications.

The implicit starting point is that 'one can only *de*-centralize what was once centralized' — that decentralization would be superfluous if latency and agency limits did not interfere. In reality, though, a decentralized concept can only be described in terms of multiple different, simultaneously valid beliefs about that concept held by independent agencies.

The feedback mechanism between them may be as simple as measuring the average or the majority, or as complex as a statistical model simulating an underlying physical process. In any case, there are two different kinds of risk to manage: *precision* and *accuracy*.

The proper measure of precision is the correspondence between an estimate and the actual measurement, a one-to-one comparison of a local follower variable and a remote leader variable. Measuring accuracy, however, requires reference to the "true" measurement, a many-to-one assessment of many remote peer variables, each of which we presume has *some* potential claim upon the truth.

To the degree that each agency's measurements of the shared concept reflect *independent* sources of error, and those values are numeric, the Central Limit Theorem establishes that the distribution of the averages approaches a Gaussian normal as the number of observations increases [63].

There are many similar statistical methods that can increase the accuracy of an estimate, depending on the probability distribution of the phenomenon being measured. In fact, if the semantics are well-known, a range of operations can be enabled within a bounded error ("balances within $5") using *escrow locks* [169].

More generally, assessment methods that can incorporate multiple agencies' opinions are strictly superior to the strategy of choosing only one agency to follow (i.e. more choices are Pareto optimal [199] — one can't be made worse off by merely making more choices available).

An ultimate expression of this principle is the constitutional right to a trial by a jury of one's peers. As stated in a recent judgment by the US Court of Appeals for

the Ninth Circuit, "fact-finding by a jury, rather than by a judge, is more likely to heighten accuracy" [216].[21]

[21] It certainly would have in that case: Allowing a judge addicted to marijuana to impose the death penalty "highlight[s] the potential risk of accuracy loss when a capital decision is reposed in a single decision-maker" [216].

Chapter 8: Estimated Systems

We have argued that neither centralized nor distributed architectural styles function at all outside the 'now horizon,' so a new approach must be devised to connect islands of locally-centralized or locally-distributed systems. The foundation of any such approach is to accepting and managing the risk of disagreement between local proxies and the remote resource.

The first step is to improve the *precision* of a local proxy: minimizing the error between the current value of the local resource and the (single) remote resource it corresponds too. Later, in the next chapter, we will also address the challenge of *accuracy*: choosing a 'true' value by assessing several remote resources.

In this chapter, we are going to introduce two new styles for working with estimated resources. The first, REST+E, is an educational example that does not actually specify any additional properties or constraints — it merely explains how REST is already intended to behave on real-world networks, where $d = \infty$. Caches, TCP sockets, passwords, and content-negotiation are all features of that help induce BASE properties for ordinary request-response interactions across the Internet.

The second style, ARREST+E is more interesting: once we can presume an event model exists, we can posit a much richer set of estimators. The essential shift is that once an architect can refer to time-series data — a stream of changing representations of the same resource over time — we can add intelligence to the infrastructure to predict current values more precisely.

8.1 Property Specification

We offered both declarative and operational definitions for estimation in §2.3.3. The former definition was stated in terms of the probability that the current local value is equivalent to the current remote value, which is ultimately bounded by the degree auto-correlation of the time series. The latter definition postulated the existence of an *estimator function* to generate such local values.

To be sure, establishing the degree of autocorrelation is domain-specific, and furthermore, does not begin to account for cross-correlation between other time series. For example, the weather report may exhibit enough of a diurnal cycle that a component that has been cut off from the network for several hours can still presume that nighttime temperatures ought to be cooler. Even more specific predictions might be possible if additional event histories were also available, such as the historical and current air pressure and wind direction.

In a pedantic sense, our definition still stands, since any domain-specific knowledge of meteorology would be reflected in the auto-correlation of the "weather report" as a whole, which subsumes these other variables. Nevertheless, we forth-

rightly disclaim that our definition takes a purely information-theoretic approach to the predictability ("entropy") of a variable.

On that basis, we can specify four mechanisms that both of our estimated styles use to induce BASE properties of representation transfers:

Minimize Latency. Best-effort networks can lose, delay, and reorder messages. Un der such conditions, retransmitting a message multiple times can only serve to lower the average latency. Similarly, using unique sequence numbers for each message can enable recipients to sort them in order and discard duplicates.

Minimize Buffering. Efficient networks cannot buffer messages indefinitely. Without buffers, senders must compress, coalesce, or discard messages to ensure that their transmission rate remains at or below the bandwidth limit of the channel. This mechanism is known, in general, as summarization.

Minimize Error. Approximate networks can keep functioning using expired information. When using out-of-date data, receivers must predict the current value based on past observations. This can be as trivial a policy as inertia — REST+E presumes that the future will be much like the present — or as complex as a time-series-driven simulation.

Ignore Spam. Self-centered networks keep functioning because recipients must be responsible for their own trust decisions. In an open, dynamic multiple-agent system, there can be no single authority for establishing trust. Instead, from the perspective of any one agency, the security challenge is to discard everything *except* messages from other agencies it trusts. (Note that this is quite separate from assessing whether the information itself is trustworthy, an aspect we will address in the next chapter.)

8.2 REST with Estimates (REST+E)

In this section, we will present an interpretation of REST's default behavior when applied to asynchronous networks. Many of the architectural elements that were guaranteed to work on finite-*d* networks can only offer some *probability* of success under such conditions.

8.2.1 Style Definition

> The REST+E architectural style induces BASE properties by constraining all representation transfers to use the `TCP/IP`, `Cache`, `AccessControl`, and `ContentNegotiation` Connectors, respectively.

In REST *per se*, the challenge of coping with consensus-freedom is either pushed down to lower layers (e.g. TCP) or ignored (e.g. cache inconsistency). The central

constraint of the REST+E style is *inertia*: past values are presumed to be current values.

For the sorts of human-readable resources found in the global hypertext/hypermedia application domain, this is not an unreasonable strategy. As the analysis of expiration policies towards the end of §5.1.1 pointed out, infrequently edited resources on networks with low median latencies can still achieve relatively high probabilities of simultaneous agreement. Of course, this becomes less tenable as we attempt to develop Internet-scale applications that manipulate resources changing at up to 5 Hz.[22]

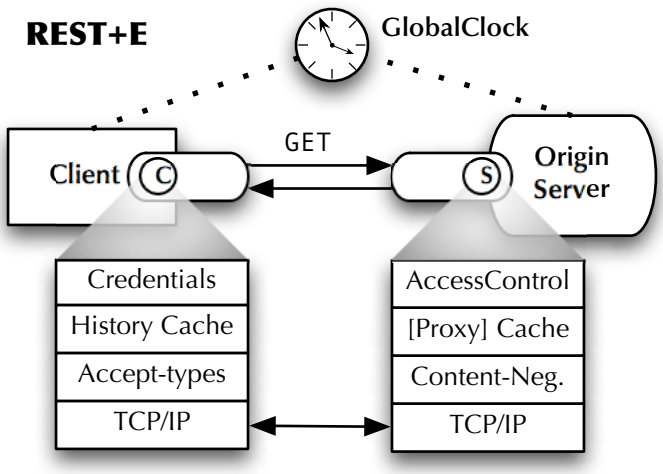

Figure 17: Illustration of the REST+E architectural style.

Within the performance envelope of the hypertext/hypermedia domain, we can identify several connectors that induce BASE properties in practice. Best-effort retransmission is handled by TCP stacks; approximation is provided by caches that return stale data; self-centered trust management is enforced by password-based access controls; and efficient use of bandwidth is made possible by negotiating appropriate media formats.

[22] A general rule of thumb is to presume that the average latency of packets traversing the public Internet is on the order of 100-200 msec. An equivalent way of stating this lower bound is "5 Hz." This includes simple presence and instant messaging, for example, but rules out 30fps videoconferencing. And indeed, the ambitions for any generic Internet-scale event notification service tend to focus around human inputs and low-frequency sensors, rather than attempting to also subsume dedicated multimedia conferencing protocols. For more on this partitioning of the range of feasible applications, see [125, 191].

8.2.1.1 The TCP Connector

Representation transfers are presumed to be reliable because they are implemented using TCP sockets. By dividing up representations in to smaller TCP *segments* for transmission, and assigning unique sequence IDs to each, the TCP stack built into Internet hosts hides the problem of minimizing *d* several layers below HTTP itself [181].

Nevertheless, TCP segment lifetimes are still limited to those of IP packets. Most implementations will thus presume a connection has been lost after only a few minutes. Thus, "best effort" representation transfers with REST+E are more accurately characterized as "best-effort for up to five minutes."[23]

8.2.1.2 The CACHE Connector

Inertia is most directly reflected by REST's caching policies. REST applications can take advantage of caching at several layers. The Web today sports caches at origin servers, for load-balancing; at proxies, for shared/public access; and even within browsers, for maintaining user navigation history. In fact, the HTTP/1.1 standard specifically states:

> By default, the Expires field does not apply to history mechanisms. If the entity is still in storage, a history mechanism should display it even if the entity has expired, unless the user has specifically configured the agent to refresh expired history documents. [68]

The EXPIRES: header that quote refers to is part of HTTP/1.1's implementation of the REST caching model. Representations have a creation DATE: and either a relative or absolute EXPIRES: deadline. They can also include an ETAG: that uniquely identifies that representation's contents for future IF-MODIFIED-SINCE: revalidation requests.

This model acknowledges risks of imprecision due to clock skew and network partitions by creating an entire vocabulary for MAX-AGE, MIN-FRESH, and so on [158]. The underlying assumption is that for sufficiently high-frequency resources, clients are expected to poll (a/k/a revalidate) often.

Approximation by inertia also applies to lock resources in WebDAV. The default behavior of an ordinary REST+D authoring tool beyond a DAV server's now horizon

[23] Since 255 seconds is the putative maximum lifetime of an IP packet. In practice, the remaining-lifetime field is often interpreted as a *hop-count* limit rather than a wall-clock limit, which expires much sooner than a full 255 seconds on modern networks. Remember, IP was designed in an era of Bell 212A modems offering mere hundreds of bits per second — and yet it still functions on today's terabit links!

is to presume that a lock remains in force until its expiry deadline. If, for whatever reason, you can't trust the administrator of the origin WebDAV server not to steal your lock; or eliminate the possibility of server failure; or if your PUT messages are delayed to arrive after your lock has expired, you still risk overwriting an interim state of the resource (a 'lost update').

8.2.1.3 The AccessControl Connector

In REST+E, real-world security risks intrude in the form of various password authorization and cryptographic privacy schemes. There is definitely an acknowledgement of agency conflict between clients and servers, but the power to control access lies *solely* with the server in REST+E.

With HTTP authentication, a UserAgent client explicitly delegates trust by revealing its credentials to the server.[24] It need not employ an additional trust manager — but only because the strict client-initiated request-response interaction model ensures that any reply message must have been a) solicited and b) originated from a trusted server.

8.2.1.4 The ContentNegotiation Connector

REST+E only models one-shot request-reply interactions, in which the reply is generated 'instantaneously' and buffered for later delivery over a limited-bandwidth link. There are no provisions for updating a transfer in progress, or for conserving bandwidth allocation across several requests ('flows,' in QoS parlance).

The latent HTTP features that *do* come to the fore to increase efficiency in REST+E are content-negotiation and content-transfer-encoding. These features help select an appropriate representation for a resource, based on client preferences and capabilities.

While negotiation is typically described in terms of selecting an appropriate file format or language/locale, it can also be used to strip down content for handheld devices or perform other types of summarization [75].

HTTP/1.1 also specifies a curious feature that crosses the boundary between the network and presentation layers: the ability to compress or otherwise re-encode an

---

[24] Literally. It remains an astonishing design flaw of the Web that the standard BASIC authentication mechanism in HTTP transmits passwords in the clear. Even cookie-based and HTML form-based analogues are rarely more secure.
The initial emphasis on statelessness in Web architecture relegated proposals for more modern security schemes such as digest-authentication [77] to the fringes, much less building in support for more complex public-key infrastructures (which in turn were swept under the rug of separate "encryption" protocols such as SSL and S-HTTP [188]).

entire transfer. It is specified in the CONTENT-ENCODING: header, which is similar to (yet entirely different from) the CONTENT-TRANSFER-ENCODING: header in the Multipurpose Internet Mail Extension specification (MIME, [78, 79]).

8.2.2 Validation

The four REST+E facilities correctly recast ACID representation transfers as BASE representation transfers by attempting to recover as much probability of agreement between the estimate and the real value as possible in a single request/response interaction.

	Goal	[Old] Elements	New Constraints	Induced Property
REST+E	Refer to a read-only centralized resource beyond its 'now horizon.'	[TCP/IP] [CACHE] [ACCESSCONTROL] [CONTENT-NEGOTIATION]	Inertia assumes that the most recent representation is still valid, until cache revalidation fails.	*Approximate agreement*. The local proxy should be in simultaneous agreement *P%* of the time.

Table 11: Summary of the REST+E architectural style.

BEST-EFFORT. The use of TCP minimizes latency caused by message loss, at least on a timescale of minutes.

APPROXIMATE. The use of caches uses an implicit prediction rule that the past value is still current. When using REST+Polling with caches, the risk of disagreement is limited to one and a half round-trip times, divided by the average lease duration.

SELF-CENTERED. The use of server-driven authentication challenges and one-shot request-reply interaction ensure that unsolicited or untrusted responses cannot be transferred.

EFFICIENT. The use of content-negotiation and compression can minimize bandwidth requirements, if not quite ensure that the total bandwidth is less than capacity.

NON-INTERFERENCE. There is a further obligation when validating the REST+E style to establish that these four new elements do not interfere with each other. Indeed, there are synergies between them: retransmission increases the probability of successful delivery of the latest information, improving precision; caching and access controls eliminate network traffic in support of efficiency; and content-negotiation minimizes bandwidth to "leave room" for TCP retransmission.

8.2.3 Implementation Issues

The implementation challenges for REST+E are well-known issues in Web implementation [132]. Cache efficiency and coherence have been thoroughly investigated [34]. Maintaining meaningful security policies with little more than access control lists and URL pathname patterns is another well-known challenge [210]. The risks of depending on offline verification of credentials has also largely been sidestepped by requiring on-line solutions for certificate revocation and electronic payments [62, 167]. Finally, content transcoding schemes for using channel capacity efficiently emerged soon after the invention of proxy servers [75] and even form the basis for commercial "Web accelerators" [57].

8.3 Asynchronous, Routed REST with Estimates (ARREST+E)

Inducing BASE properties using time-varying event sources has much greater potential than basing estimates on one-shot request/reply interactions. In ARREST+E, we will add several types of estimating connectors that strengthen retransmission, summarization, prediction, and trust management by discerning patterns over time.

8.3.1 Style Definition

> The ARREST+E architectural style induces BASE properties by constraining all representation transfers to use the StoreAndForward, Summarizer, Predictor, and TrustManager Connectors, respectively.

Estimators are a variant of connectors that do not ignore expired representations/notifications, but use a Web of Trust and statistical, economic, or other risk-management techniques to approximate the current value of a remote resource. This allows Estimator connectors to emit "synthetic messages" on their output interfaces even when there may be too much or too little information arriving on their input interfaces.

Specifically, our approach for extending REST into a consensus-free style requires eliminating references to remote resources. If all components refer solely to local resources; and estimators are the only connectors that can exchange BASE representations between local and remote resources; then we can neatly isolate the effort of measuring a remote resource's value *precisely*.

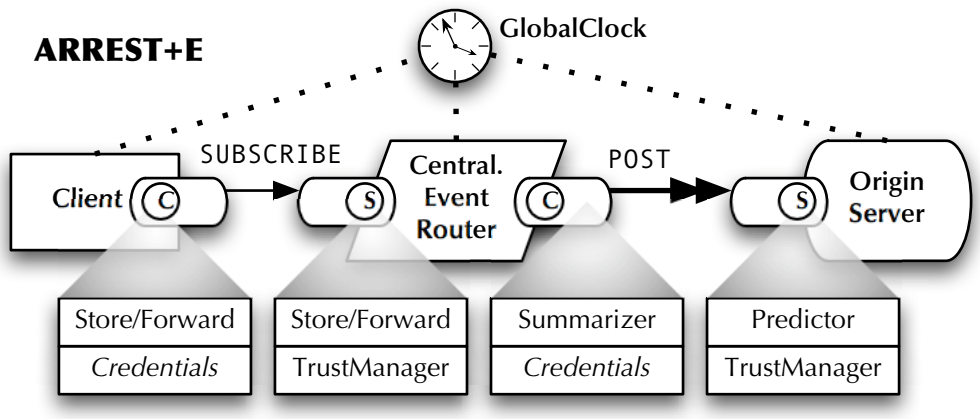

Figure 18: Illustration of the ARREST+E architectural style.

8.3.1.1 The StoreAndForward Connector

The goal of the StoreAndForward connector is to increase the probability of successful representation transfer. Primarily, this requires attempting retransmission for as long as the representation remains valid — not merely giving up after IP times out.

Retransmission requires some mechanism for distinguishing unique messages from repeated ones. The longer a message can remain pending, the larger the set of identifiers must be. Of course, duplicate detection also places a correspondingly larger burden on communication endpoints to maintain longer-lived cache of previously-seen IDs.

All of this suggests that StoreAndForward requires a longer-lived address space than TCP's (IP address, port number, sequence number) triples. After all, while the network layer may be limited to routing ephemeral IP packets, at the application layer we routinely encounter event notifications that may remain valid for decades — consider the permanence of Message-ID: headers for Usenet articles or email messages.

ISO Global Unique Identifiers (GUID, [110]) would be one choice for identifying representations. We propose reusing the mechanism already used for caching in HTTP: entity tags (ETag:). Typically generated by applying a one-way hash function to the representation body, entity tags allow recipients to efficiently detect and discard duplicate representations. This builds upon the similar role we proposed for entity tags in §5.2.2.1, namely, suppressing routing loops for a CentralizedEventRouter.

A secondary function for identifiers is to indicate relative ordering, as with TCP's increasing sequence numbers for each segment. Our second assumption for StoreAndForward is that every event notification has a creation timestamp. This still permits some endpoints to recover an ordering, to the degree that resources are

available to do so (buffer space, processor time, etc). This is separate from timestamps that may be found *within* an event notification to allow application-specific reordering. The Data Stream Management System (DSMS, [15]) research community has also encountered this distinction, which they termed intrinsic vs. extrinsic timestamping.

8.3.1.2 The SUMMARIZER Connnector

Since, in practice, latency can be exacerbated by buffering due to bandwidth limits, there are real advantages to having a connector to suppress transmission of expired messages, to say nothing of coalescing related sequences. Such a SUMMARIZER adds yet another requirement for identifying event notifications, in this case a *name* for the source of each event.

In conjunction with the entity tag and creation date, the resource identifier included in each REST-style representation allows a SUMMARIZER to correlate changes to an event source over time. Unlike a messaging system, where each email or article is presumed to be an atomic, write-once resource that must be delivered intact or not at all, event notifications are only intended to be snapshots of a single phenomenon over time.

Therefore, while messaging services treat their payloads as unique and anonymous, event notification services are entitled to "mess with the message." A simple example is the role of Timewarp's anti-messages [114, 141] or Usenet cancellations [109]. A later "message" can annihilate an earlier one, given the correct credentials and an identifier to locate it. Another example is how modern GUI environments manage input events for "slow" applications. The GUI can coalesce individual keystrokes, cancel out temporary depressions of a shift key, or even elide intermediate mouse movements by replacing several traces with straight-line motion to the latest coordinates.

8.3.1.3 The PREDICTOR Connector

The goal of the PREDICTOR is to synthesize new data from old. Our original specification was narrowly stated in terms of the auto-correlation of a time-series, but a PREDICTOR also gives the architect freedom to divine correlations to any other available information or models. That is because it encapsulates a Turing-complete program rather than only a statistical function.

This definition permits prediction strategies ranging from the generic (frequency-counting) to domain-specific (weather forecasting). Recall that for REST+E, we derived that even the most basic strategy, inertia, can provide high reliability if a resource changes slowly enough. Unfortunately for the architect, it is often precisely when a resource is updated that the user's fortunes may be most affected — literally, in the case of financial market data.

Thus, in ARREST+E, an architect is free to deploy more complex strategies that take full advantage of event streams. However, a PREDICTOR must conserve its use of both space and execution time, since it may be executed every time new information arrives. Furthermore, the theoretical requirement for bounded space and time is compounded by the practical requirement that many such processes are executing concurrently. The challenge of scheduling large-grained, highly-stateful prediction processes within an event router may begin to resemble process scheduling in an operating system, or query planning within a database management system.

8.3.1.4 The TRUSTMANAGER Connector

The fundamental new challenge for enforcing the Self-centered property, by comparison to REST+E, is that adding asynchrony and routing permits spam. The *notify()* interface can be invoked by any other component at any time, unlike the traditional receive-reply mechanism.

By contrast, in REST+E, messages can only arrive if they is specifically authorized: servers only accept requests from trusted clients, and TCP sessions only return data from that server. Now that we cannot defer message integrity to the network layer, we posit a TRUSTMANAGER for enforcing integrity at each endpoint.

Formally, a mechanism for inducing multilateral extensibility could be reduced to a filter that verifies that a trust relationship exists between any sender and recipient components' owners, whether before transmission or after receipt. We can imagine even stronger forms of this requirement that also tag each message with access rights as a form of capability-based security, but that is not required in our basic model of trust.[25]

An even more sophisticated complication is granting and revoking trust dynamically. While ordinary REST request-reply interactions last for only a short time, a subscription relationship may persist for much longer. This raises the risk that the Web of Trust may change during the subscription, as when passwords are lost or payments fall into arrears. A TRUSTMANAGER provides a locus of control for an architect to specify more detailed policies, potentially down to the level of approving or disapproving each transmission over the network.

Indeed, with some knowledge of application semantics, it is conceivable that a TRUSTMANAGER could synthesize compensating messages that can cancel out the input

[25] In other words, if Alice trusts Bob, our simple model requires trusting Bob not to go publish her secrets in the newspaper. Email works like this today; there is no control of over message forwarding. One could imagine, however, a stronger mechanism that would still require later recipients to ask Alice for decryption keys, say — if Bob were willing to cooperate.

of participants discovered to be untrustworthy later on — perhaps using the same mechanism as the cancel-messages discussed above for SUMMARIZERS.

8.3.2 Validation

The four ARREST+E facilities correctly recast ACID representation transfers as BASE representation transfers by attempting to recover as much probability of agreement between the estimate and the real value as possible by taking advantage of available information about the past values of a variable.

BEST-EFFORT. Since STOREANDFORWARD minimizes latency, it induces the property of Best-effort representation transfers. Identifying stored messages uniquely allows architects using ARREST+E to continue retransmitting messages as long as they remain valid; indeed, to even transmit already-expired messages if bandwidth is available. This allows the application to tolerate loss of network connectivity between agencies for far longer than TCP/IP can alone.

	Goal	New Elements	New Constraints	Induced Property
ARREST+E	Refer to a read/write resource connected by a faulty network beyond its 'now horizon.'	STOREANDFORWARD Connector that adds end-to-end retransmission and acknowledgement policies.	End-to-end retransmission of notifications and acknowledgments.	*Best-Effort data transfer*: Cope with message loss.
		PREDICTOR Connector for encapsulating Turing-complete prediction functions of past states.	Predict probable current state from past data (when possible).	*Approximate estimates*: Cope with message delay.
		TRUSTMANAGER Connector that drops notifications from untrusted sources.	Ensure that all reachable endpoints are also trusted.	*Self-Centered*: Cope with dynamic participation.
		SUMMARIZER Connector to resample queued events at lower frequency.	Enforce bandwidth limits; Prohibit transmission of superceded data.	*Efficient data transfer*: Cope with network congestion.

Table 12: Summary of the ARREST+E architectural style.

Approximate. Since Predictors minimize error, they induce the property of Approximate representation transfers. Using both past values of a variable and other information about the state of the application, a Turing-complete Predictor can achieve lower risk of disagreement than Caches can alone.

Self-Centered. Since TrustManager discards "spam," it induces the property of Self-centered representation transfers. Because event notification (ARREST) permits unsolicited messages to arrive at any time, it is not sufficient to simply assume that ever reply message was initiated by a request to a trusted server, as AccessControl does alone.

Efficient. Since Summarizers minimize buffering, they induce the property of Ef ficient representation transfers. By processing entire sequences of queued messages representing past values of a variable, a Summarizer can compress or cancel several messages more effectively than one-shot ContentNegotiation can alone.

Non-Interference. There is a further obligation when validating the ARREST+E style to establish that these four new elements do not interfere with each other. Beyond the synergies identified earlier in §8.2.2, these connectors are even more effective because they can update the local proxy resource ('event source') at any time. A StoreAndForward connector could take advantage of multiple paths/protocols to deliver the same message, as long as those paths exist in the recipient's Web of Trust. A sender's Summarizer connector could take advantage of the behavior of the recipients' corresponding Predictor connector to minimize errors.

8.3.3 Implementation Issues

Only StoreAndForward and TrustManager could be expected to be completely independent of event source semantics, and hence implemented in a generic ARREST+E framework. There are decent default examples of Summarizers suitable for generic implementation — dropping outdated 'in-flight' representations, concatenation, compression — but domain-specific coalescing could do much better. In contrast, Predictors must be entirely user-supplied to do better than the default presumption of inertia.

8.3.3.1 Retransmission Strategy

StoreAndForward requires a concrete retransmission strategy — it cannot simply flood a channel with more and more copies of a message in hopes that one will get though. The schedule could be paced by an acknowledgement counting, sliding window protocol to offer reliable, ordered service much like TCP (Zack is a sample implementation of such an end-to-end protocol, see §10.2.3.2).

Or, it could take advantage of entirely different communications technology, such as one-way satellite paging [98]. Even un-acknowledged radio broadcasts can be

effective at reducing the probable latency, especially when using appropriate forward-error-correction and multi-scale, multi-resolution rebroadcast policies.[26]

Finally, a retransmission schedule also needs to take into account the performance characteristics of the underlying transfer protocol. It may even have to incorporate mechanisms to estimate current bandwidth, delay, and jitter limits. This way, an event router can distinguish between acceptable retransmission frequencies for an instant-messaging interface on the LAN, or an email interface over a dial-up modem on the WAN. Of course, this performance data is also relevant for choosing summarization strategies, too.

8.3.3.2 Impedance Matching

A potentially more sophisticated approach is to combine pending notifications to form a single, larger-grain summary. In some GUI toolkits, this is known as coalescing mouse positions or keystrokes [87]. In the larger context of knowledge management, Gelernter memorably described resampling as the "squish" operation [93]. User preferences also govern this process; it was first termed 'impedance matching' in the 'Getting to Know' notification service (GtK, [183]) for controlling the pace of collaborative activities. An even more sophisticated model for comparing and merging partial sequences of collaborative actions to a shared document is operational transforms (dOPT, [61]).

For the architect, all of these approaches require carefully designing data structures suitable for incremental materialization. For example, in our Web browser-based sample applications, we developed a pattern for using hash tables keyed off of an event's `KN_ID` to reassemble a wide variety of data structures. In the `INTROSPECT` topic browser, a directory listing is not a `GET` snapshot in time, but a `WATCH`ed resource that collects entries one at a time, so the list is always dynamic (for better or

[26] One intriguing possibility is the use of structured namespaces to coordinate multiple variants of the "same" resource. Suppose one wished to broadcast a weather report to soldiers in the field over a one-way channel. A naïve approach would be to rebroadcast a weather map as often as possible and hope that soldiers gained access to the network for long enough to receive a complete copy. A more sophisticated approach might be to fill the channel with, say, 10km^2-scale maps half the time, and 1km^2-scale the rest. This increases the odds that during a short burst of connectivity, a soldier recovers at least *some* of the map, and the effective resolution could increase over time. See [206] for a similar approach applied to 3D scene rendering at variable levels of details.
It would be quite intriguing to apply such abstractions to non-graphical data as well; imagine reconstituting a newspaper by subscribing to separate `HEADLINE`, `STORY`, and `PHOTO` event sources...

worse). This turns out to be another example of multi-resolution agreement — if we had infinite bandwidth, we'd just send the entire updated directory listing on each change, saving the client this sort of effort. Instead, we needed to offer the tradeoff of delta-coding finer-grained updates at finer-timescales.

8.3.3.3 Stateful PREDICTORs

It is beyond the bounds of this thesis to discuss the vast array of techniques for predicting the behavior of event sources; the literature of control theory in applied mathematics is but one starting point [207]. Our focus is on describing how such techniques are packaged as software components and employed within our architectural styles.

The most challenging aspect is devising a common programming interface that supports the entire range of estimators, from the trivial null estimator (inertia) to weather-prediction supercomputers. At the highest level, it is simple enough — a PREDICTOR connector receives representations of external resources on its input and produces representations of a local proxy resource on its output — but the internal framework support could be more elaborate. How should past values be stored for later reference? How much? Are time-series a basic data type? Moreover, there is the (insoluble) challenge of ensuring that PREDICTORs terminate.

8.3.3.4 Credential Management

Enforcing a Web of Trust for each agency complicates the trust management problem significantly. Essentially, credentials that were once centralized in a single ORIGINSERVER or USERAGENT must now be distributed across multiple agencies — and using estimates for them entails further risks.

Not only are these risks of construction (buffer overflows, cryptographic weakness, etc), there are risks inherent in decentralization as participants join and leave the application. The least of these are traditional challenges of managing a Public-Key Infrastructure (PKI, [128]). The novel aspects are addressed in the growing literature on decentralized trust management (PolicyMaker, [28]) and distributed security infrastructures [192].

Chapter 9: Decentralized Systems

In this chapter, we shift from the challenge of precision to that of *accuracy* — from estimating facts to assessing opinions. All of our previous architectural styles have manipulated resources owned by a single agency; this is the first time we explicitly address agency conflicts — the possibility that the agencies that own the components of an inter-organizational application do not trust each other.

Consider how a currency-trading application would cope with the decentralized market for foreign exchange. In the absence of any "true" value for an exchange rate, a weakly-consensus-based approach would be to "fork" a decentralized resource into multiple, independent resources and simply attempt to estimate each one individually.

Suppose a foreign exchange trader has access to two banks. In that case, the hypothetical resource `TRADER.COM/YEN-RATE` could be replaced by the set {`CHASE.COM/JPY`, `CITI.COM/JAPAN`}. However, that leaves the user with two seemingly-unrelated choices: ChaseYen and CitiYen. Both would end up being listed alongside Euros and UK Pounds as though they were pseudo-currencies of their own.

If the program specification is to trade dollars whenever the Yen-rate for dollars exceeds the Euro-rate, how could a fair comparison be made? After all, the customer's ultimate requirement is for legal tender Japanese Yen, not ChaseYen or CitiYen — though both banks' resources happen to be simultaneously valid estimates of it.

Decentralized styles of software architecture need to accommodate some mechanism for recovering a *single* local belief based on multiple external inputs. For one customer's purposes, their experience is that Chase is twice as reliable as Citibank, so their model should set `TRADER.COM/YEN-RATE` := $\frac{2}{3}$ `CHASE.COM/JPY` + $\frac{1}{3}$ `CITI.COM/JAPAN`. For another, it may be to choose whichever bank's offer is cheaper, depending on whether the application needs to buy or sell Yen overall. More sophisticated economic models could then be applied in order to make such *assessments*.

9.1 Asynchronous, Routed REST with Estimates and Decentralized Decisions (ARRESTED)

This is the last of our new styles, in which we graduate from approximate agreement with a putative single valid value, to complete consensus-freedom with multiple, simultaneously valid values.

The new capability we propose adding to ARREST+E is a component that determines the local value of a decentralized concept by assessing a panel of other

agencies' corresponding resources. This sort of Decentralized decision function serves to share control of a resource among several agencies, just like the other decision functions in our distributed styles (REST+D and ARREST+D). Unlike those two, however, a decentralized solution cannot be based on transactions, locks, or other attempts to enforce ACID properties.

9.1.1 Property Specification

Inducing the property of consensus-freedom is difficult because it requires proving a negative: the absence of consensus requirements. Indeed, inducing this property reveals a paradox: the freedom to disagree can only be secured by agreement over concepts that are even more fundamental. A troop of scouts, each with their own compass, may disagree over what direction the troop is headed — but only because there is still universal agreement over the meaning of North, South, East, and West.

All an architectural style can do is help architects to locate and eliminate dependencies on external agencies' resources. Our proposed mechanism for eliminating consensus requirements is to isolate such dependency within an assessment function and a correspondence function, as defined in §2.3.3 and §2.3.4, respectively.

9.1.1.1 Assessment

A more direct example of a quintessentially decentralized algorithm is the *consistent hash* [117]. An ordinary hash function maps an infinite domain onto a finite range; a cryptographic ("one-way") hash also frustrates attempts to invert that mapping as well. As long as two agencies agree to use the same hash function (and any parameters, such as a secret key), simultaneous agreement over its output is assured — even at a distance. Essentially, the algorithm is centralized, even if the inputs are private.

The difficulty arises when some of its parameters are allowed to vary. For example, a common use for hash functions is to distribute items across a set of containers. For several independent components to share the same containers (e.g. a pool of Web cache servers), they must ensure that they all map items the same way. This requires simultaneous agreement among all participants over the hash algorithm *and* the set of available containers.

To add or subtract a cache from that pool, the entire system would have to be halted first — two clients with inconsistent views of the pool would no longer share the same mapping of items to containers. A better solution would permit clients with *different* views of the set of containers to still map the same items to *approximately* the same places.

Informally, a consistent hash function addresses this problem by generating a series of probable containers to rendezvous at, rather than just one. Thus, even when clients have different snapshots of the cache pool membership at different times,

they can still be assured that they will rendezvous at the same place with arbitrarily high probability if they read or write to enough of those candidate containers.

The broader lesson of this solution is that this is an example of a "shared-nothing" architecture: individual agents can still calculate the decentralized concept "which container does this item belong in?" without requiring simultaneous agreement with *any* other agency.

Another related formalism is Herlihy's wait-free hierarchy of synchronization primitives [106]. A concurrent data object is wait-free if it does not busy-wait nor block waiting for some other condition to become true. Such objects can be characterized by their *consensus number*, the maximum number of processes that it can solve the consensus problem for without waiting. In that terminology, an assessment function must not rely on any synchronization primitive with a consensus number >1.

9.1.1.2 Correspondence

Of course, independence without cooperation leads to isolation, not decentralization. There must still be a way for agents to establish that other, external resources also represent the "same" concept. This requires agreement over syntax, semantics, and trust relationships; in our definition of decentralization in §2.3.4, these requirements appeared in the guise of the *correspondents()* function.

In the caching example, there is an "obvious" layer of syntactic agreement; in this case over the alphabet for URL strings that identify items and the particular consistent hash function. The next layer of abstraction would be concerned with establishing consistent semantics: ensuring that the result of an older `GET` request is not substituted for another client's `POST` invocation. And finally, there is issue of whether a user trusts the organization running the caching proxy; consider the case of a Chinese citizen and the "Great Firewall" [166], which censors access to news and websites the government finds objectionable regardless of users' trust decisions.

9.1.2 Style Definition

> The ARRESTED architectural style induces consensus-freedom by constraining all components to replace references to external agencies' resources with an `ASSESSOR` component that incorporates information from multiple agencies without depending on consensus with them.

The specific mechanism we propose adding to our architectural styles to "eliminate dependencies on external agencies' resources" is an `ASSESSOR` component. Rather than relying on a local proxy resource to reflect the current value of a remote one — as we have in all of our previous styles — a local assessment reflects the current values held by several agencies about the same *concept*.

As long as ASSESSORS use decision functions that do not block, the result is a shared-nothing (decentralized) application architecture. This new component *also* subsumes the role of the *correspondents()* function. An architect must provide rules for an ASSESSOR to select resources that are syntactically and semantically equivalent. (Note that we already presume that the TRUSTMANAGER introduced for ARREST+E that ensures all connections in the architecture are trusted.)

By comparing Figure 19 to the ARREST+D style depicted in Figure 13, the former 3x3 crossbar has fragmented into three different ASSESSORs for each agent. The Web of Trust on the side explains which agents trust each other; so in the case of Agent #2, we can see that it relies solely upon its own local judgment. Nonetheless, both of the other Agents still choose to trust #2's values as an input to their *own* assessments.

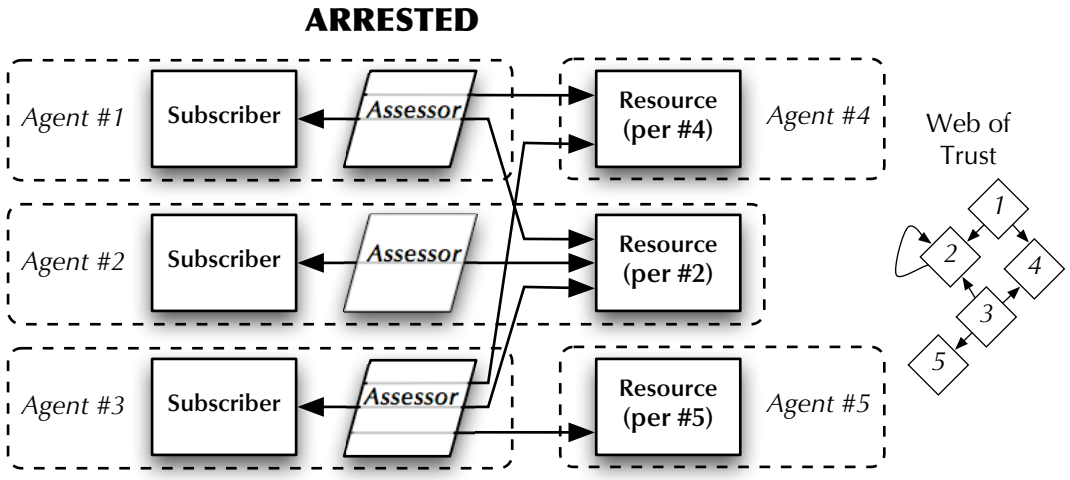

Figure 19: Illustrated example of a concept shared by 5 agencies in the ARRESTED architectural style.

9.1.2.1 The ASSESSOR Component

The unique characteristic of an ASSESSOR is that it *combines* several inputs into one output. This is a significant shift from the era of application-integration for distributed systems using brokers [172]. The essential behavior of a request broker is a *lookup()* function that resolves names to addresses in an entirely application-independent manner. This only permits brokers to *select* one input to connect a local proxy to.

Consider a purchasing application that needs to reorder paint when supplies are low. A centralized solution would be to connect the local variable PAINTPRICE directly to a remote variable maintained by PAINTSELLER-1 (or, if it were unavailable, redirect requests to PAINTSELLER-2). A distributed solution would be to connect the local variable to a marketplace website, say, PAINTBROKER. At any

given moment, PAINTBROKER would select the best price available from all *N* PAINTSELLERS.

By contrast, a decentralized solution would be to tie the value of the local variable to *all* of the suppliers simultaneously. The price the purchasing-planning application should use could be based on the best price to prevail in the market for the given color, quantity, and delivery schedule required. An ASSESSOR may therefore need to take into account supplier reliability, timeliness, and past volume discounts to determine its decision.

In the context of our earlier caching example, a consistent hash function would also be represented by an ASSESSOR, since it would "listen" to several peers' best estimates to generate its own assessment of the cache pool's current membership.

Other examples of non-blocking decision functions eligible for use in an ASSESSOR include taking the minimum, maximum, mode, median, or mean of the set of corresponding values. These simple statistical functions allow multiple-agency systems to continue operating with approximately equivalent values even in the face of "lost updates."

9.1.3 Validation

Adding decentralized decisions to ARREST+E induces consensus-freedom by replacing all references to remote resources with an ASSESSOR component that does not block trying to establish consensus with any other component.

Without a testable, declarative specification for a mechanism to enforce consensus-freedom, it is not possible to validate ARRESTED in the same manner as our other proposed styles. Instead, we have an operation definition that calls for an assessment function, *af()*, and a procedure for determining which resources are sufficiently "similar", *correspondents()*. Since ASSESSORS implement those two functions (by definition), as long as all remote references are replaced with assessments, there cannot be any scope for requiring consensus with any external agency.

	Goal	New Elements	New Constraints	Induced Property
ARRESTED	Decentralize control of a shared resource across disjoint 'now horizons.'	ASSESSOR Component that manages the risk of inter-agency disagreement over the 'true' value using a panel of opinions.	Eliminate reliable references to remote resources; only contingent estimates remain.	*Consensus-freedom:* must not presume feasibility of consensus at all.

Table 13: Summary of the ARRESTED architectural style.

ASSESSORs also do not interfere with the properties already established for ARREST+E. The inputs to an assessment function can equally well be values known with certainty or estimates with error bars. The subtler challenges of interference are not between styles, but between different types of resources.

On one hand, our consensus-free styles for estimated and decentralized resources automatically reduce to simple centralized and distributed resources if the network latency is low enough and the Web of Trust is a complete graph. There is no inherent computational overhead for ARREST+E because the ESTIMATORs won't be invoked unless representations expire in transit or bandwidth limits are exceeded. Similarly, an ASSESSOR can still implement a probabilistic locking scheme that selects a leader's value for the group.

On the other hand, if some components of the application *do* fall outside of each other's now horizons, we face the scepter of upgrading centralized or distributed resources into estimated or decentralized replacements. Imagine using a calendaring application that automatically worked with certainty at the office, but still let the user schedule probable meetings while off-line. How can these types of resources be combined or compared consistently when the user returns to the office?

The problem is that components beyond the now horizon essentially must 'secede' from the rest of the application to continue operating. After that point, the sum of decentralized variable and a centralized variable yields another decentralized variable, thus "infecting" the rest of the system. A more familiar, and formal, way to depict this is a lattice of security levels — how even public-domain information incorporated into a 'top secret' report becomes classified, too [18].

The major implication of this line of research is that declassification — or, in our case, re-establishing consensus once network conditions and credentials permit — requires careful consideration, and possibly even human supervision [58]. Alternatively, architects will have to consider isolating aspects of their applications that are consensus-based from those that are consensus-free.

9.1.4 Implementation Issues

As with ESTIMATORs, we do not have enough implementation experience with ASSESSORs to recommend an internal architecture for such functions. One intriguing model for evaluating contingent offers from multiple agencies is the 'option ladder' [30]. Just as options on stocks represent traders' assessments of prices an equity at different points in the future, different paint suppliers' bids could be considered options to buy or sell paint. For non-numeric data, it may at least suffice to enumerate various agencies' possible alternative values and determine one's own probability distribution across those.

Such speculation aside, the more significant implementation issue for ARRESTED-style applications is determining correspondence. Even our syntax for

delimiting an agency boundary could be considered suspect, since domain names alone do not designate authority over an URL — consider the case of 'personal' home pages within a departmental web site. Establishing syntactic and semantic equivalence is a significant challenge on the Web today, and even on into the promised Semantic Web of the future [24].

The two banks in our example at the beginning of this chapter used rather different URLs to identify the resources they claimed represented the decentralized concept "price of Japanese Yen." It is difficult to discern a strictly deterministic rule for establishing such equivalence, without also positing extensive common knowledge such as "domain-names-that-belong-to-banks" and "abbreviations-for-national-currencies."

Nevertheless, for some significant applications, literal pathname matching has proven workable in practice. As a matter of convention, any Web server can publish a `/ROBOTS.TXT` resource to control its use by non-human user agents, or a `/FAVICO.ICO` resource to provide an image for a site's bookmarks. In other cases, international organizations have struggled to establish practically-universal vocabularies, such as ISBN codes or airport location codes.

In our own implementations, we ended up relying on syntactic pathname-equivalence: we cluster topics by dropping the domain name, port number, and (for http: and https:) the scheme as well. The rest of the "ontology problem" was represented by routes. For example, we had several newspaper headline feeds flowing into topics organized by domain name; to create an orthogonal classification by topic, we manually created routes from, say, `/WSJ.COM/HEADLINES` to `/FINANCE`. Patterns also emerged for hierarchical taxonomies: by organizing headlines by date in the form `/YEAR/MONTH/DAY/HOUR`, and creating routes back "up" from each `HOUR` directory to each `DAY` directory, and so on, subscribers could specify an appropriate timescale to fine-tune their queries (subscriptions).

Chapter 10: Infrastructure

In this chapter, it falls to us to establish that our new architectural styles are indeed implementable — without limiting the scalability or range of applications built in those styles. Our primary evidence for validating these claims is found in the Mod_PubSub open-source project, which adds an event notification service to the Apache Web server [126].

However, our investigation has been iterative, since we have developed new insights about our styles and their relationships only by extending and applying Mod_PubSub. In particular, its development predates our insight that there are four separate types of resources (centralized, distributed, estimated, and distributed). As a result, this chapter is organized into two portions: Mod_PubSub's design and implementation; and an analysis of the feasibility of each new architectural style.

10.1 The Mod_PubSub Project

In the summer of 1998, we convened the Workshop on Internet-Scale Event Notification (WISEN) at UC Irvine, followed by some informal "birds-of-a-feather" sessions at the fall Internet Engineering Task Force (IETF) meeting. These efforts culminated in our earliest proposals for adding event notification to the Web [125]. The common thread to the various experiments we have undertaken with Internet-scale event notification is our vision for a "Two-Way Web": pairing HTTP client and server functionality at each node to form a simple, effective, and familiar foundation for peer-to-peer architectural styles [97, 122]. Later, we expanded our approach to embrace a broader concept called Application-Layer Internetworking (ALIN, [123]) that casts network messages between applications as convertible to a hypothetical intermediate format analogous to IP, a generic Transfer Protocol (TP).

We have learned quite a bit from developing an experimental event-routing infrastructure along these lines over the past five years. Our original prototype implementations were released separately as an open-source project in November 2002 called Mod_PubSub (by analogy to the Apache Mod_* naming convention for Web server extensions) [126]. Eventually, these experiments became the foundation for the award-winning commercial edition sold by KnowNow, Inc [220].

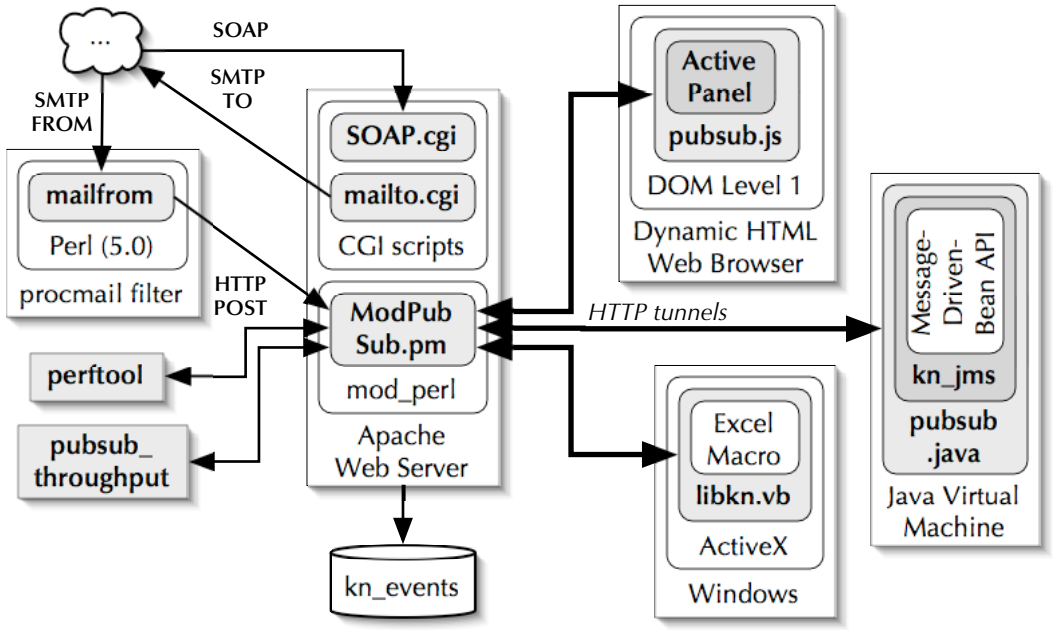

Figure 20: Components of MOD_PUBSUB operate within several other systems.

In this subsection, we will proceed to expand upon our approach to, design for, and implementation of an event notification service built on top of existing REST-style infrastructure.

10.1.1 Approach: Application-Layer InterNetworking (ALIN)

Rather than approximating the ideal of a single event notification 'bus' connecting all components of an architecture, we aim to implement a decentralized event router: a network element that can be owned by independent agencies yet still interoperate without presuming shared information about topics, subscribers, or events. The advantages of this approach correspond to those of connecting multiple agencies' independent networks in an internetwork rather than attempting to build a single, shared network: independent administration; improved reliability, availability, and scalability; and interoperability with a wider range of protocols.

By viewing event notification as a *routing* problem, we have mapped the challenge of reliable, publish-and-subscribe messaging at the application layer down to the precedents set by Layer 3 internetworking (*viz.,* IP and UDP). Unlike prior generations of LAN-scale messaging systems, which merely exploited that metaphor by relying on reliable multicast UDP at Layer 3, our effort begins by positing a new Layer 7 abstraction, the Transfer Protocol (TP). TP represents a lowest-common-denominator protocol, more or less, for HTTP, FTP, SMTP, NNTP, and other Internet application-layer protocols, as well as messaging systems such as IBM's

MQSeries, Tibco's Rendezvous, Microsoft's BizTalk and myriad proprietary JMS implementations.

Roughly speaking, an "IP datagram" corresponds to a TP message: IP's 32-bit addresses grow into topic name URLs; IP's fixed-length payloads become arbitrary; IP's untyped (binary) payloads become richly typed; IP's per-packet security becomes per-message security; and so on. Originally, IP was grudgingly tolerated as only a protocol for 'networks of networks,' insufficient to replace LAN protocols inside organizations – just as SOAP [31] is viewed today. Eventually, though, IP became the dominant Layer 3 protocol throughout the computer industry, on LANs as well as WANs. We suspect there is similar potential for standardizing communications between components of a software application.[27]

One reason we are attracted to the ALIN approach is that it leads to some novel ways to describe the properties induced by an architectural style. For example, if we can see that the only path between two networks crosses through a firewall, we can immediately grasp that the only data flowing between them must be trustworthy in the eyes of the firewall's owner. Similarly, one can add security policies for restricting access to selected hosts/components by deploying single-sign-on appliances as bottlenecks. One can also improve reliability and capacity by adding multiple links between subnetworks. In general, architectural diagrams can simultaneously be viewed as routing graphs and vice versa. This allows us to describe new functionality by showing that all routes between two points must go "through" the components that implement those constraints.

10.1.2 Design

The design we chose for extending Web tools to support event notification reflects the symmetry of the ALIN approach: building a "Two-Way Web" where data can flow asynchronously between peers, not just from servers back to clients. The implementations in the next subsection are based on Web design precedents for its protocols, event model, routers, microservers, and applications.

10.1.2.1 Protocol

The ideal Web event notification protocol would be a direct implementation of the `SUBSCRIBE` method we posited when describing ARREST. It indicates a request to "let `<CALLBACK-URL>` 'watch' all the traffic sent to this `<REQUEST-URL>`, by proxying such requests onward to it." This is a deliberate and appropriate extension

[27] Rose's Blocks Extensible Exchange Protocol (BEEP, neé BXXP, [194]) is a related approach that came out of the late-90's interest in a unified application-layer protocol toolkit [118]. Unlike TP, though, BEEP delves much deeper into connection-management to provide finer-grained APIs.

of HTTP because it directly builds upon its underlying hypertext model, addressing well-known concerns about using HTTP as a substrate raised in [162].

In practice, though, we can't modify Web browsers and Web servers to support our new method. It has become notoriously difficult to advance the deployed versions of HTTP [130, 131]. For convenience, we then mapped our ideal form into a set of parameter names and values to use with `X-WWW-FORM-URLENCODED` marshalling in ordinary `POST` requests. Any HTTP/1.1 field was imported using the `DO_` prefix; any private, experimental field not yet harmonized with HTTP was relegated to `KN_`. Thus, method names, maximum ages and such parameters of `SUBSCRIBE` are now `DO_`'s, but our non-standard Expiry field, which only accepts integer positive seconds, is still only a `KN_EXPIRES`.

Finally, three additional, practical hurdles force us to reverse the physical flow of messages over a single TCP connection. Since we want to send asynchronous inbound IP traffic back to a desktop PC, we must contend with 1) browser scripting security limitations that limit communications only to the same or subordinate domain name as served the page; 2) firewalls that prevent externally-initiated traffic; and, most perniciously, 3) Network Address Translators (NATs, [209]), which make it impossible to identify unique ("routable") IP addresses for clients.

10.1.2.2 Event Model

In the development of packet internetworking, all communications could be reduced to a single packet format. Unlike IP, though, our event router uses a slightly richer abstraction of the information it must process: a persistent identifier for each event notification, including the topic that contains it.

Consider that "an email in a mailbox" is by no means a mere transcript of the original SMTP network traffic. In fact, it took decades from there to the innovation of Internet Message Access Protocol (IMAP, [53]), which *does* posit a richer storage and security model for identifying emails and mailboxes. Similarly, our router is not merely transcribing and reformatting bursts of HTTP traffic, but instead models events and topics, respectively, as resources and collections per the WebDAV specification [230].

Because one original interpretation of WebDAV's goal is "turning a website into a filesystem," many of its semantics are derived from experience with many prior filesharing protocols. This gave us a simple yet flexible model for event contents: files in directories represent different representations of the same event source ('topic'). More importantly, every event has a persistent, unique name ("event id" or `KN_ID`) that is used for updating events previously-published events and suppressing routing loops.

Our design goes even further by following the motto that "everything is a topic or an event" to support reflection on metadata on events, topics, and subscriptions.

Unlike WebDAV's additional unique methods to retrieve (PROPFIND) and modify (PROPPATCH) resource metadata— who created it, when, and so on — we chose to fold many of these items into the format of the event itself. Therefore, we added KN_ fields to report an event notification's last-modified time, expiry time, purported publisher, the last route it traversed, and so on.

The second type of reflection includes similar information for topics: who created it, when, its maximum resource usage limits, degree of durability it should be stored with, and so on. Not surprisingly, we chose to represent topics *as* events, albeit within a special container: a KN_SUBTOPICS topic within every topic. This makes it reasonable to discover new topics by ordinary Web crawling, as well as eliminating the need for special methods to create, update, or delete routes — just work with the corresponding entry in the *parent* topic's KN_SUBTOPICS.

Our third sort of reflection extends this solution to capture the state of the routing table itself: Who's listening? Who's allowed to listen? Once again, every topic has a ./KN_ROUTES subtopic that contain representations of every subscription from that topic. For administrators to learn what routes exist, they need only subscribe to that topic to be notified of changes to the routing table. Naturally, such inspection also creates a ./KN_ROUTES/KN_ROUTES subdirectory, and so on. Access controls applied to each level can be used to express more complex trust management policies (see §10.2.3.3).

10.1.2.3 Router

The basic interface in and out of our routers is an HTTP/S POST request. Our other supported protocols, such as FTP file sensors or outbound email alerts, are gatewayed through this HTTP interface. Any MIME payload or HTML FORM element is acceptable as the body of an event (KN_PAYLOAD). Here is the basic proce dure for processing incoming requests:

First, we rely on the built-in access controls of the host Web server's security to ensure this was a legitimate request. If so, the router takes care of automatically filling in certain headers like the creation date, expiry, and KN_ID.

Next, the router determines whether this event is "new news." If this event is identical in every material way to the event currently stored under that KN_ID, processing stops. Otherwise, it is added to our repository. In the current implementation, this is a transactional commit; the router does not reply with 200 OK until the event is written to stable storage.

If it is a request to create a subscription (publish into a KN_ROUTES topic), a new route is created. If that new route request has a DO_MAX_AGE or DO_MAX_N option, it looks in its cache to replay as much past event notification traffic as it may have stored so the new subscriber can 'catch up.'

Continuing as we would for any ordinary event notification, the router queues it up for onward delivery to each of the eligible destinations found in that event's . /KN_ROUTES subdirectory. At that point, we check whether it can be sent (security), where it should be sent (including running it through a filter service, optionally), what format it should be sent in (e.g. XML or JavaScript), and how it should be sent (direct copy to tunnel socket or forwarding to another, external router?).

Note that currently, MOD_PUBSUB uses a nested delivery loop. This leads to a transitive closure (a depth-first search of the entire routing graph, in the worst case) and requires every notification to be completed before the router can release the publisher. Such reliability is a risky assumption, especially because it ought to be best-effort, only forwarding events as spare processing power permits.

10.1.2.4 Microserver

Completing this design requires a modification to the communication pattern between USERAGENTS and ORIGINSERVERS to permit sending and receiving multiple responses. Clearly, an ordinary REST SERVER connector is already prepared to accept asynchronous notifications. Client-side support is enabled by the ASYNCSERVER and ASYNCCLIENT connector types. Unlike the ordinary SERVER and CLIENT connectors, the ASYNC extensions hold open a network connection for the duration of a WATCH relationship. This approach is known as the 'Two-Way Web' since it essentially pairs a REST CLIENT and SERVER together. It shifts the discourse from encapsulating notifications as multiple-responses (sent to a client) towards multiple-requests (sent to a server) — hence our coinage of the term "microserver."

While one could use any handy HTTP library to access a router, that is not the same as providing a complete Internet-scale event notification framework. A microserver is the whole kit of platform-specific code provided to developers. In general, a complete microserver does at least three things: hook into the "local" event model; manage stable queues for reliable pub/sub; and manage interactions with the network layer.

The first aspect of microserver design is a 'personality' issue, in the sense popularized by the Mach microkernel's emulation layers for other UNIX interfaces [5]. For example, the JavaScript microserver not only provides generic PUB(), SUB(), and UNSUB() calls, but in conjunction with LIBFORM or KN_HTML, it also can hook directly into the browser's GUI event loop as specified by the Document Object Model (DOM, [233]). This allows the model-view-controller architectural style [129] to be applied naturally to previously static Web pages. Similarly, in Excel we took pains to add ON_EVENT hooks that call users' own macros when events arrive, making event-driven development more familiar to power users. In Java, we provide both low-level queuing and Java Message Service-compliant (JMS, [113]) interfaces.

The second aspect of microserver design concerns reliability. We claim `PUB()`, `SUB()`, and `UNSUB()` are non-blocking calls. This means we maintain working queues for pending inbound and outbound traffic. For applications where it is critical that an event presumed published indeed will be (to a best-effort) local stable storage is necessary.

Finally, of course, is the network layer. The microserver controls the two TCP/IP connections to and from the router. Wherever possible, we use "native" libraries for connecting to the Web. Part of the beauty of its ubiquity is that far more man-years have been invested in optimizing, say, Microsoft Windows' `WININET`, than any home-grown HTTP stack might.

10.1.2.5 Applications

In addition to the event routing infrastructure *per se*, the `MOD_PUBSUB` package also includes a set of test scripts, test applications, widgets, and a sophisticated router administration tool called `INTROSPECT`. Figure 21 is a screenshot of the developer tools and samples included with a recent release of `MOD_PUBSUB`.

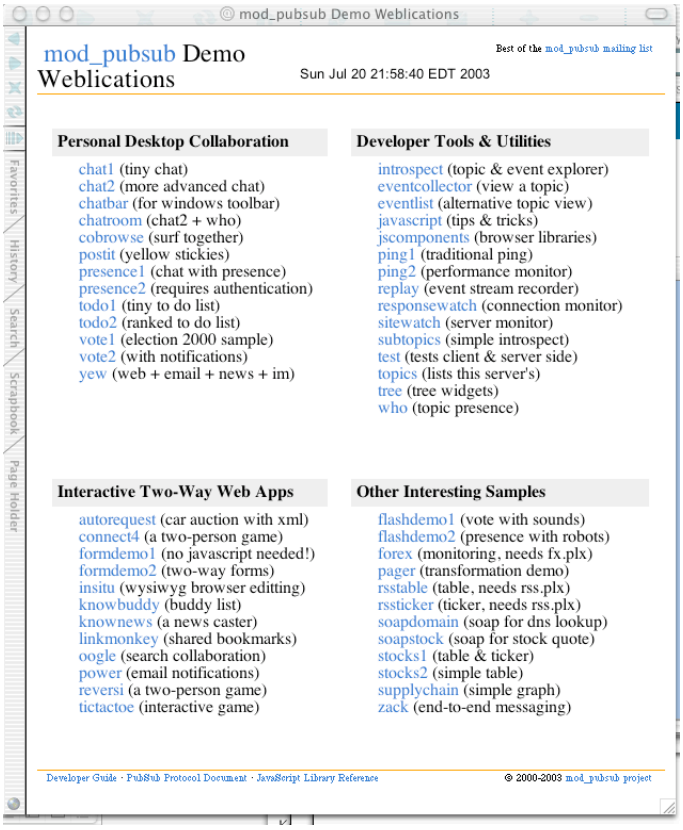

Figure 21: A listing of sample applications in the `MOD_PUBSUB` distribution.

The client and server test harness is a client-side JavaScript application that invokes a corresponding series of some thirty-odd server-side CGI scripts that publish

and subscribe to the router, and then concludes with some additional, browser-specific tests. Since the project was initially managed according to the philosophy of "extreme programming" [17], it is significant that the server tests were developed alongside the original `PUBSUB.CGI` code itself. For each new proposed router feature, a corresponding test was written before implementation. Therefore, even with a very small development team, we were able to construct an independently-developed test suite, rather than just adding diagnostic debugging information to the router under development. This approach proved its worth when the same test programs were applied to later commercial and Python editions of the router.

Along the way, we also developed a set of common components for application developers. The majority are specific user interface elements for Web browsers — bar charts, Miller columns, HTML parsers and validators — but a few are interesting examples of server extensions. For example, the `PAGER` application invokes a server-side script to "compress" text using a table of abbreviations (e.g. "Los Angeles" becomes "LAX"). By simply subscribing via the `PGRSPK.CGI` "filter," any application can take advantage of this event transformation.

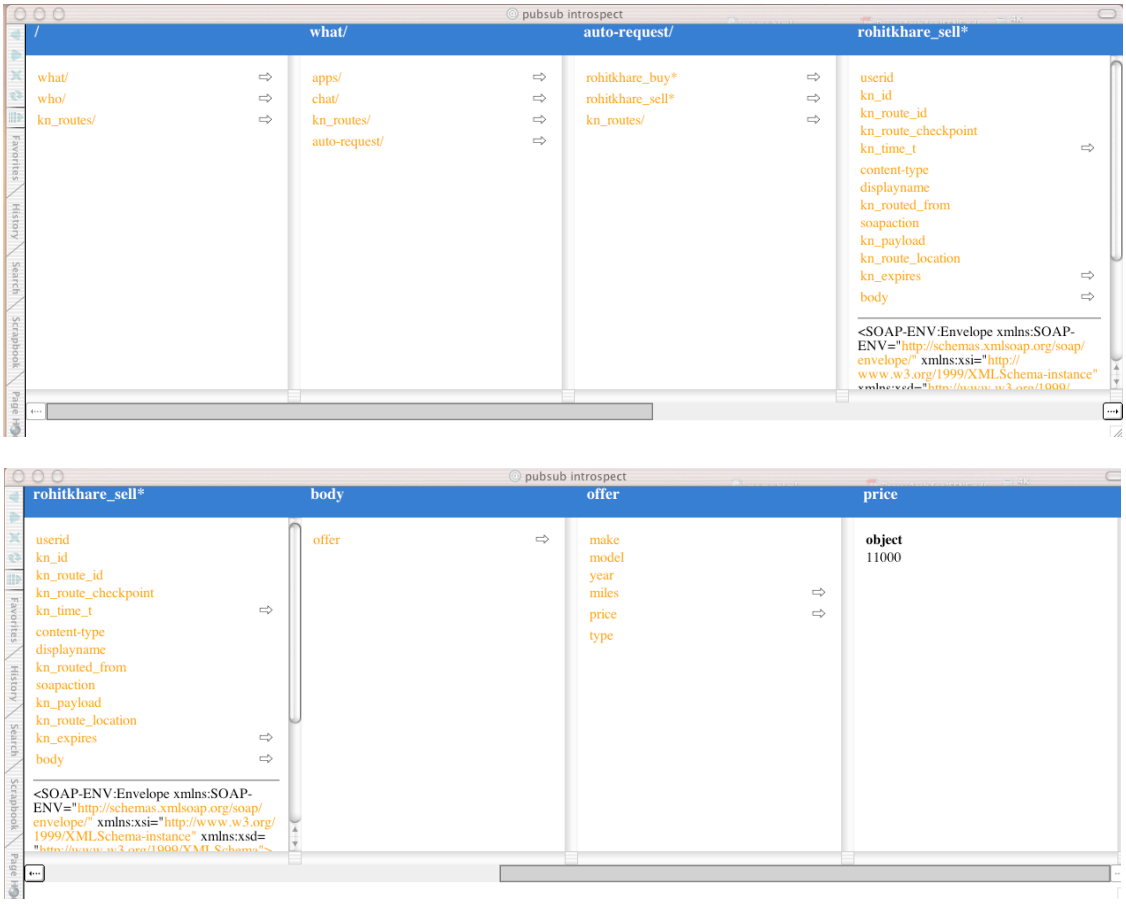

Figure 22: The `INTROSPECT` application examining a for-sale event notification.

The INTROSPECT application is designed for administering an event router; commercial versions even include an online editor for the router's own configuration files and credentials. As shown in Figure 22, it takes advantage of many of our new widgets and libraries. It presents a browseable 'topic hierarchy' by subscribing to `./KN_SUBTOPICS`. It even continues that metaphor by browsing right into the headers and values of an individual event notification; in this particular case of an XML message, it even browses the JavaScript object constructed from its body. The entire user experience is "live," so elements on display are modified as soon as new information arrives.

10.1.3 Implementations

With a Web-centric design in hand, our informal justification for building yet another Internet-scale event notification service was to replicate the ease of use and deployment of the Web. HTTP servers support decentralized hypertext by serving up a portion of the global URL namespace without any *a priori* coordination with any other Web servers, clients, or proxies; we had the same goals for a small, simple server for decentralized event routing.

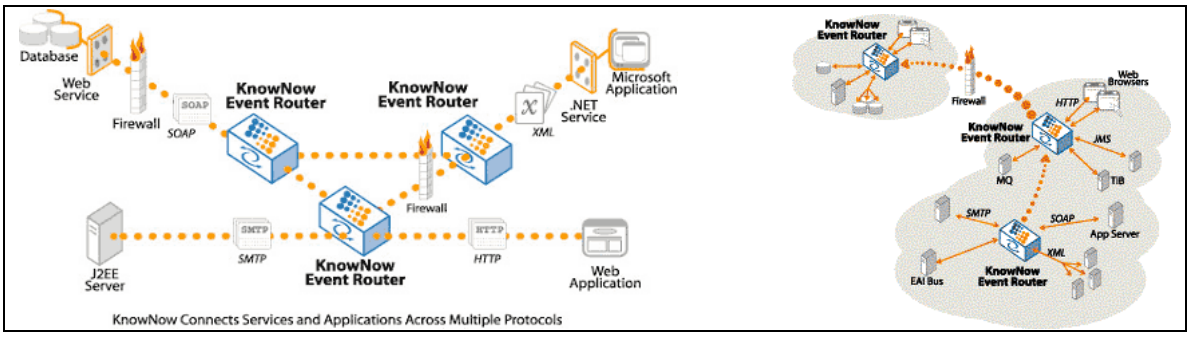

Figure 23: A multi-protocol event router can interconnect multiple organizations. (©2003, KnowNow Inc.)

Figure 23 sketches the wide range of programs necessary to develop working experimental systems. To take full advantage of existing REST-style infrastructure, these implementation efforts were dispersed across a wide range of systems, as extensions written in a range of different programming languages.

On the left-hand side is a sample deployment across three organizations. Even if placed inside or outside of a network firewall, each agency's router can establish connections for forwarding event notifications, forming an application-layer overlay network. On the right-hand side are two organizations, but the larger one includes an internal network of routers as well.

In either case, though, the business processes at hand impact databases, custom applications, Web browser-based and GUI-based interfaces, and a variety of other

legacy MOM systems. For example, many Web sites today are set up to send offers and alerts via email; by using an SMTP gateway, an existing J2EE application merely needs to be configured to send emails to the router rather than an end-user. Alternatively, a Java developer may choose to send notifications to other Java components using the JMS API; the commercial product includes a Java program that provides such an API, but translates all traffic into standard HTTP messages to communicate with the router. Similar adaptors have been developed for Tibco's Rendezvous bus and IBM's WebSphere MQ.

The broader lesson of Figure 23 is the potential for an intermediate event-notification protocol to dramatically simplify the burden of integrating 'integrated' applications. Whether by adding the adaptor to the router core, or to the client program, the essential benefit of this solution is that developers need not change their native messaging interfaces to take advantage of an interoperable protocol.

In the following sections, we will discuss both our open-source prototype and commercial production versions of this architecture.

10.1.3.1 Open Source

As Table 14 documents, MOD_PUBSUB is an extensive project, implemented in a wide variety of programming languages and runtime environments. Broadly, however, the various pieces fall into four categories: routers ("servers"), microservers ("client libraries"), sensors ("gateways"), and sample applications.

The flagship event router is `PUBSUB.CGI`, which can run within virtually any web server, but also provides additional performance in conjunction with Apache. The other key router is a monolithic, single-threaded server written in Python. Not only is `PUBSUB.PY` much higher performance, it is the more 'readable' version as well, because the control flow did not need to inverted to fit into an enclosing Web server.

Microservers, which both queue outbound messages and maintain tunnel connections to receive inbound ones, are implemented in a wider range of languages. There are even multiple implementations in the same language, because the key is API support. As with the commercial edition, there is work underway on both a simple Java library and more complex JMS provider.

Sensors are programs written to collect information from other news sites, stock quote feeds, and even other protocols (e.g. email into a topic).

Using MOD_PUBSUB with existing web tools and browsers, web pages can respond to incoming events from the network — in real-time; in plain text, HTML, XML, or SOAP format; and without relying on Java or ActiveX plugins of any kind. It works with Mozilla, Netscape Navigator, and Microsoft Internet Explorer browsers and any Web server with support for CGI scripts written in Perl [228].

Item	Description
Makefile	Makefile to create PUBSUB.JS from PUBSUB_RAW.JS, the framework and tools in the JS_PUBSUB directory, and a properly-permissioned KN_EVENTS/ directory. Use "MAKE ALL" to create, and "MAKE CLEAN" to remove made files.
LICENSE	Our BSD-style license.
kn_docs/	Documentation for this distribution, including Frequently Asked Questions (FAQ), Developer Guide, PubSub Protocol Document, JavaScript PubSub Library Reference, and JavaScript Options, Tricks and Debugging Guide.
cgi-bin/	1. **Perl PubSub Server:** A mod_perl-based Apache module (MODPUBSUB.PM). 2. Some useful CGI scripts, including email notification (FORM2SMTP.CGI), PubSub Server regression test (PUBSUB_TEST.CGI), a sample SOAP service proxy (SOAP_FILTER.CGI), and some sample transformation scripts. 3. **Perl PubSub Client Library** (PUBSUB). It is single-threaded and partially non-blocking (tunnel is non-blocking, other requests block). 4. **SOAP Gateway** (PUBSUBSERVICE.CGI). A preliminary SOAP interface to mod_pubsub. When complete, the SOAP interface will include support for incoming SOAP requests, off-host SOAP routes, and external SOAP service invocation. (SOAP design)
perl_pubsub/	The **Perl PubSub Client Library** and **Perl PubSub Server** reside in the cgi-bin/ directory. Some sample apps that use the Client Library are CHAT, LIST_EVENTS, LIST_ROUTES, and LIST_SUBTOPICS in the KN_TOOLS/ directory, and some sensors in the KN_SENSE directory.
js_pubsub/	A directory of utilities and libraries that offer a framework for JavaScript components, including sample applications, docs, and tests.
kn_apps/	A directory of **PubSub-enabled Web Browser samples**. Includes several component JavaScript libraries, such as **JavaScript PubSub Client Library** (PUBSUB_RAW.JS) that is single-threaded and non-blocking; and sets of helper libraries KN_LIB and JSCOMPONENTS.
flash_pubsub/	Some early research into a **Flash and ActionScript PubSub Client Library** (PUBSUB.AS).
php_pubsub/	**PubSub Client Library for PHP** (PUBSUBLIB.PHP), including a Publish example (PUBLISH.PHP -- works in PHP3 and PHP4) and a server-side Subscribe example (SUBSCRIBE.PHP -- only works in PHP4 with Sockets extension, and uses the EVENTLOOP.PHP and PIPEFITTING.PHP libraries for network I/O.
ruby_pubsub/	**Ruby PubSub Client Library** (LIBKN.RB). It is multi-threaded and blocking.
python_pubsub/	1. **Python PubSub Server** (PUBSUB.PY) that runs standalone from the command line. It is our goal to have the functionality of PUBSUB.PY match that of CGI-BIN/PUBSUB/MODPUBSUB.PM so we have two reference implementations. 2. Other helper Python libraries.

Table 14: An inventory of files in the MOD_PUBSUB project.

Item	Description
python_pubsub/ *(cont'd…)*	3. **Python PubSub Client Library** (PUBSUBLIB.PY). It is single-threaded and non-blocking. There are also some **sample apps** that use this library: CHAT.PY, PUBLISH.PY, SUBSCRIBE.PY, and REPEATER.PY. Works on Python 2.1+ for Windows, and Python 1.5+ for Linux, Solaris, and Mac OS X. 4. An alternative **Python PubSub Client Library** (LIBKN.PY) that is multi-threaded and blocking.
java_pubsub/	**Java PubSub Client Library** (client and chat app). It is multi-threaded and blocking.
cxx_pubsub/	**C++ PubSub Client Library** (LIBKN) that can be compiled on Windows or PocketPC. It is multi-threaded and blocking. There are versions for Visual Studio 6 and Visual Studio.NET; also included are sample apps such as an RSS publishing program, and documentation that can be built using DOXYGEN.
c_pubsub/	**ANSI C PubSub Client Library** (LIBKN) that can be compiled on any flavor of Unix. It is single-threaded and partially non-blocking (tunnel is non-blocking, other requests block).
kn_sense/	Sample sensors for scraping data from websites and publishing them to a PubSub Server. For example, see the RSS sensor (RSS.PLX).
kn_tools/	Useful ancillary utilities, including: 1. Command-line chat using the Perl PubSub Client Library (CHAT.PLX). 2. Command-line tools using the Perl PubSub Client Library (LIST_EVENTS.PLX, LIST_ROUTES.PLX, and LIST_SUBTOPICS.PLX). 3. Command-line ping using the C PubSub Client Library (PUBSUB_PING). 4. Command-line throughput performance testing tool in ANSI C (PUBSUB_THROUGHPUT). 5. Command-line performance testing tool in ANSI C (PERFTOOL). 6. Command-line throughput performance testing tool in Java (PERFTEST_THROUGHPUT.JAVA). 7. JavaScript object serializer (JS_OBJECT_SERIALIZER.HTML). 8. JavaScript compressor (JS_COMPRESS.SH). 9. JavaScript localizer (KN_LOCALIZE.SH). 10. JavaScript character set browser (CHARAT). 11. Some useful JavaScript bookmarklets.
push_manager/	A connection pooler for use with CGI-BIN/PUBSUB/MODPUBSUB.PM to enable it to serve thousands of simultaneous connections. Note that this program does not currently work with most platforms (derived from thttpd).

Table 16: *(continued)*

MOD_PUBSUB includes a set of JavaScript libraries that can be included in Dynamic HTML pages. These files make available a set of objects and functions that you use to subscribe, unsubscribe and send events. In addition, these scripts can identify the current user. Getting the current user's identity requires both that the PUBSUB.CGI

instance is hosted on a password-protected site; and that the JavaScript files are generated by PUBSUB.CGI. All of this can be done with a single <SCRIPT> tag which points to the MOD_PUBSUB instance:

```
<!-- PubSub JavaScript Library for Live 2-Way Forms-->
<script src="../../cgi-bin/pubsub.cgi?
do_method=lib2form"></script>
<!-- Check it out, a two-line program! -->
   <form action="kn:/what/chat/">
      <input name="kn_payload" size="20" /></form>
```

Program 20: A sample JavaScript MOD_PUBSUB chat application.

The PUBSUB.JS JavaScript helper libraries also include support for sending and receiving SOAP-formatted XML messages. Converting received SOAP-formatted XML messages into JavaScript objects makes accessing the properties within it much simpler, as well as strongly-typed (integers, booleans, strings, and so on) [72].

10.1.3.2 Proprietary

KnowNow's commercial version is largely compatible with MOD_PUBSUB, but its internal architecture is tuned for much greater performance and reliability. Rather than plugging into an external Web server as a Perl CGI script, KnowNow LiveServer™ is a complete event router for Windows, Solaris, and Linux written in C++ that contains an embedded Web server of its own (AOLServer, [10]).

Rather than using the filesystem as an event store and polling it for changes, LiveServer uses an embedded BerkeleyDB [175] cache and a threaded connection pool. Rather than process each event against each subscription individually and using external scripts for content filtering, it can load parsers and query processors into the kernel and cache parsed events to boost throughput.

It is also designed for tighter integration with security solutions, including SSL support and integration with outboard SSL hardware accelerators. Using CONNECT to tunnel SSL also ensures compatibility by circumventing proxy caches and other network elements that might interfere with microserver tunnels [140]. It also provides an authentication API to check credentials against external directories.

LiveServer also supports up to 8-way clustering through the simple expedient of replication by cross-routing. Any event notification sent to one cluster member is relayed to the others before being acknowledged. Because of the reflection built into the design, merely forwarding every event publication is sufficient to mirror the entire router's state — even the creation of subtopics or modification of configura-

tion parameters is caused by an event notification. Since writes still block until replicated on all cluster members, though, this only increases notification (read) throughput.

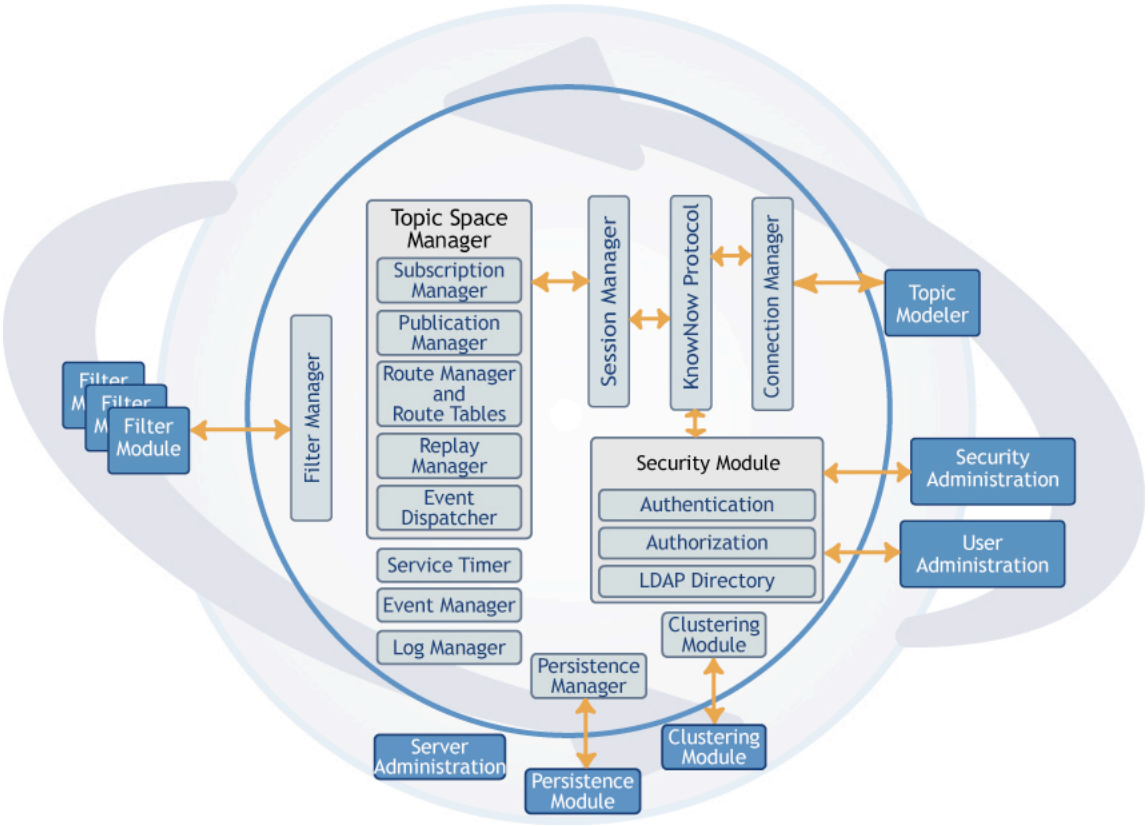

Figure 24: A block diagram of the internal architecture of the commercial router. (©2003, KnowNow Inc.)

10.2 Feasibility of Our New Styles

Having described our infrastructure, in this section we will show that it can be used in demonstrations of the feasibility of each of our new styles. It suffices to show that each new component, connector, and constraint we have posited have, in fact, been realized in some manner in our routers, microservers, and sample applications. Since our implementation predates our styles, it is also important to show that the various extensions work together, rather than interfere.

We will proceed in a slightly different order than we originally derived our styles in the last few chapters. We will begin with our baseline style, REST; add each of our four basic types of functionality to it (asynchrony, A+REST; routing, R+REST; estimates, REST+E; and decisions, REST+D); and then combine those facilities to support centralized (ARREST), distributed, (ARREST+D), estimated (ARREST+E), and decentralized (ARRESTED) resources.

10.2.1 Existing Elements

Our specific intention in designing MOD_PUBSUB as an extension to Apache was to leverage as many of the benefits of REST as possible. Because of this, our obligation to validate the feasibility of REST and REST+P only extends to defining the specific new elements we added to the original definition of REST in §4.2.2.

10.2.1.1 Synchronization (REST)

All of our experiments with MOD_PUBSUB presumed that GLOBALCLOCK was implemented by the host operating system(s) our components were executing upon. This was reasonable because the smallest unit of time our system observed events at was only one second (such low-resolution measurement of time was a consequence of using the filesystem to communicate between PUBSUB.CGI processes). The clock skews we observed while using Simple Network Time Protocol (SNTP, [154]) synchronization were much less than one second.

We did not account for malfunctions or malfeasance. For example, we avoided incorrectly-set locales (time zones) by opting to store and compare times in seconds-since-UNIX-epoch format whenever possible. Similarly, we attempted to mitigate timestamp fraud by ensuring that routers always updated creation timestamps, KN_TIME_T, according to their own clocks, not the clients'.

Freshness was the only constraint we defined in REST, by using a GLOBALCLOCK to enforce expiration deadlines. We enforced this lazily: whenever an event was read from disk, it was immediately deleted if it had already expired (see EVENTFORMAT.PM). Note that this still does not prevent exhaustion of storage resources: events that omit a KN_EXPIRES header remain valid indefinitely.

10.2.1.2 Polling (REST+P)

Because REST+P is a *cul de sac* on our path towards deriving a decentralized version of REST, it should not be surprising that we did not implement it in our project directly. Nonetheless, polling is already the most prevalent way to add pseudo-"real-time" information to Web sites: the "META-REFRESH" facility instructs Web browsers (USERAGENTs) to reissue the current request after a specified interval (see §4.3.4). This transformed an ordinary USERAGENT into a REST+P POLLINGCLIENT.

Since MOD_PUBSUB also allows direct GET access to topics and individual events, developers could configure it to add such a META-REFRESH header to the HTML representations of directory listings and files it returned. Nevertheless, this simple POLLINGCLIENT is much less powerful than using STOREANDFORWARD to retrieve specific ranges of events, including use of checkpoints (see §10.2.3.3).

10.2.2 New Elements

The next step towards deriving a decentralized version of REST was identifying four basic types of functionality to add on to it. Once again, it is worth noting that our development of MOD_PUBSUB predates our understanding that these were four orthogonal facilities. There is not necessarily a one-to-one correspondence between our new components and connectors and the internal design of MOD_PUBSUB. Rather, the various aspects of the MOD_PUBSUB project — along with services provided by the operating system and underlying Web server — can be combined and configured to support each element of our new styles.

10.2.2.1 Asynchrony (A+REST)

A NOTIFYINGORIGINSERVER is obliged to transmit additional replies whenever a resource's representation changes, for the duration of the client's WATCH request. This is much *less* sophisticated than implementing a full-scale publish/subscribe system — recall that the concept of SUBSCRIPTIONS will not arise until ARREST (see §10.2.3.1).

One of our earliest experiments was to hold open a connection to a browser and stream down to browsers isochronously (KNSERV.PL, deprecated in [126]). We found that, with sufficient padding to flush internal buffers, incremental rendering of Dynamic HTML in modern browsers could enable us to trigger alerts or modify the displayed document without dropping the connection. We dubbed this technique "isochronous HTTP," in part to differentiate it from the better-known usage of the term "persistent connections" to refer to pipelining of HTTP requests [157], but also to indicate that messages would arrive at the client with the same *relative* delays as the changes at the server.

Extending this result to support asynchronous notification of resource changes required an implementation of the WATCH method. We accomplished this in PUBSUB.CGI by, first, formalizing the definition of a TUNNEL to send multiple responses encapsulated within a single HTTP/1.1 200 OK response; and second, by implementing an event-observation loop that monitored changes to a representation on the filesystem at 1Hz (since file modification times are only measured in seconds in UNIX).

10.2.2.2 Routing (R+REST)

A ROUTINGPROXY is obliged to redirect its reply to another component, as specified by the client. The simplest implementation would be to preserve as much of the REST model (and existing implementations) as possible. For that reason, we rewrite a response message as a new request message intended for the destination URL. The details of such rewriting largely follow the rules for proxying requests in HTTP/1.1.

This behavior is implemented by content transformation: before a reply message is to be delivered, the KN_CONTENT_TRANSFORM header can specify another proxy server it should be filtered 'through', with the output of that process replacing the original reply.

However, this remains a *nested* call, meaning that not only does the intermediate server add latency when transferring back a reply, the agency that owns it must also be trusted not to modify the representation when doing so. In practice, architects using MOD_PUBSUB can use SUBSCRIPTIONs to "dropbox" topics to achieve the same effect in the ARREST style.

10.2.2.3 Delegated Decisions (REST+D)

The role of a MUTEXLOCK is fulfilled by filesystem locks, albeit indirectly. The goal of the REST+D architectural style is to achieve pairwise ACID agreement between one client and one server. MOD_PUBSUB relies on underlying locking mechanisms in the operating system to enforce Atomicity, Isolation, and Durability.

ATOMICITY. Buffering incoming requests until the entire body is received (according the stated CONTENT-LENGTH: of the request message) ensures atomicity by preventing partial, corrupted event publication. This relies on locking of independent temporary files by concurrent PUBSUB.CGI processes running within the same Web server.

ISOLATION. Clients wishing isolation from other publications on the same topic can simply omit an event's KN_ID field. That way, the PUBSUB.CGI process receiving it will generate a unique name for it. This is effectively using the topic's directory listing as a test-and-set atomic object. Admittedly, this does not enforce isolation for concurrent writes to the same *event,* only for the same topic.

DURABILITY. Finally, durability of a write is indicated by the publication of a status event. Whatever the particular level of durability the router administrator chooses — tape drive or RAM disk or anything else — a PUBSUB.CGI process can only generate a 200 OK reply after the write is completed. Note that the status information can itself be represented as an event, and routed to another component using the KN_STATUS_TO header.

To reduce the possibility of a lost-update further, there MOD_PUBSUB includes another experimental router called PUBSUB.PY, written in Python [225].[28] It enforces serialization by handling *all* requests to publish or subscribe in a single thread (rather than in concurrently-executing CGI scripts). In the commercial edition of the router, an embedded BerkeleyDB is used to enforce transactional integrity [175].

[28] Arguably, it is the clearest, most readable implementation in the whole project — and it's much faster than PUBSUB.CGI, even with MOD_PERL acceleration.

10.2.2.4 Estimation (REST+E)

As we noted during our discussion of REST+E in §8.2.1, its implementation reflects current practice on the Web. TCP/IP stacks built into modern operating systems take care of retransmitting and re-ordering data to survive network outages of several minutes. Access controls built into modern Web serves can protect resources based on users, passwords, and pathname patterns, using a wide range of cryptographic mechanisms. Content negotiation is less consistently supported on the Web, but it typically offers automated selection of languages, file formats, and file size limits. Web caching, in browsers and in proxy servers, can already return stale representations of a resource when disconnected from the network.

All of these facilities come into play automatically when browsing event sources (topics) or retrieving event representations from a MOD_PUBSUB implementation (as long as the portion of the filesystem used to store events is itself accessible through standard GET requests). Even for POST requests that publish new events, the destination URL (KN_TO) must be re-validated using the user's own credentials (since every user is always accessing the 'same' /PUBSUB.CGI resource, automatic path-based discrimination is not effective in this case — the relevant information is actually encoded in one of the input parameters).

10.2.3 Composite Elements

We can now proceed to addressing the specific requirements of centralized, distributed, estimated, and decentralized resources by combining the four basic facilities in ARREST, ARREST+D, ARREST+E, and ARRESTED, respectively.

10.2.3.1 Event Routing (ARREST)

The key distinction between a CENTRALIZEDEVENTROUTER and the NOTIFYINGORIGINSERVER and ROUTINGPROXY it subsumes is the appearance of first-class SUBSCRIPTION resources. In line with our emphasis on supporting reflection in MOD_PUBSUB, this distinction takes the form of KN_ROUTES subtopics for every topic. That is, the 'routing table' of all outstanding subscriptions with the same source can itself be considered an event source, generating notifications for creation or modification of new entries within it.

The behavior of a CENTRALIZEDEVENTROUTER is constrained by the same event loop described for routers in §10.1.2.3. Reflection also makes it easy to explain how subscriptions are later destroyed. Judicious choices of access controls on recursively nested KN_ROUTES subdirectories also allowed us to reduce Access Control Lists (ACLs, e.g. [163]) for publish and subscribe rights into publication rights alone (since subscription is the side-effect of publishing into the KN_ROUTES subtopic). Further-

more, by permitting SUBSCRIPTIONs to be independent events, even 3rd and 4th parties can be authorized to connect components in the ARREST style.

10.2.3.2 Distributed Decisions (ARREST+D)

While the existence of distributed WebDAV repositories using locks to manage synchronization shows promise (e.g. Subversion, [51]), we pursued an alternative strategy for implementing FAIRMUTEXLOCK for ARREST+D. The Bakery algorithm pseudocode provided in Program 19 can be implemented end-to-end (without relying on a central router for arbitration).

However, on the real Internet, the average message latency is much less than the maximum *(d)*. Therefore, we combined the basic queuing mechanism with an acknowledgement-counting protocol to block until delivery was assured. This experiment, ZACK, is shown in Figure 25. Note that its correct operation *does* assume reliable, ordered delivery of network messages eventually; it merely avoids presuming that the central router is the sole transaction manager.

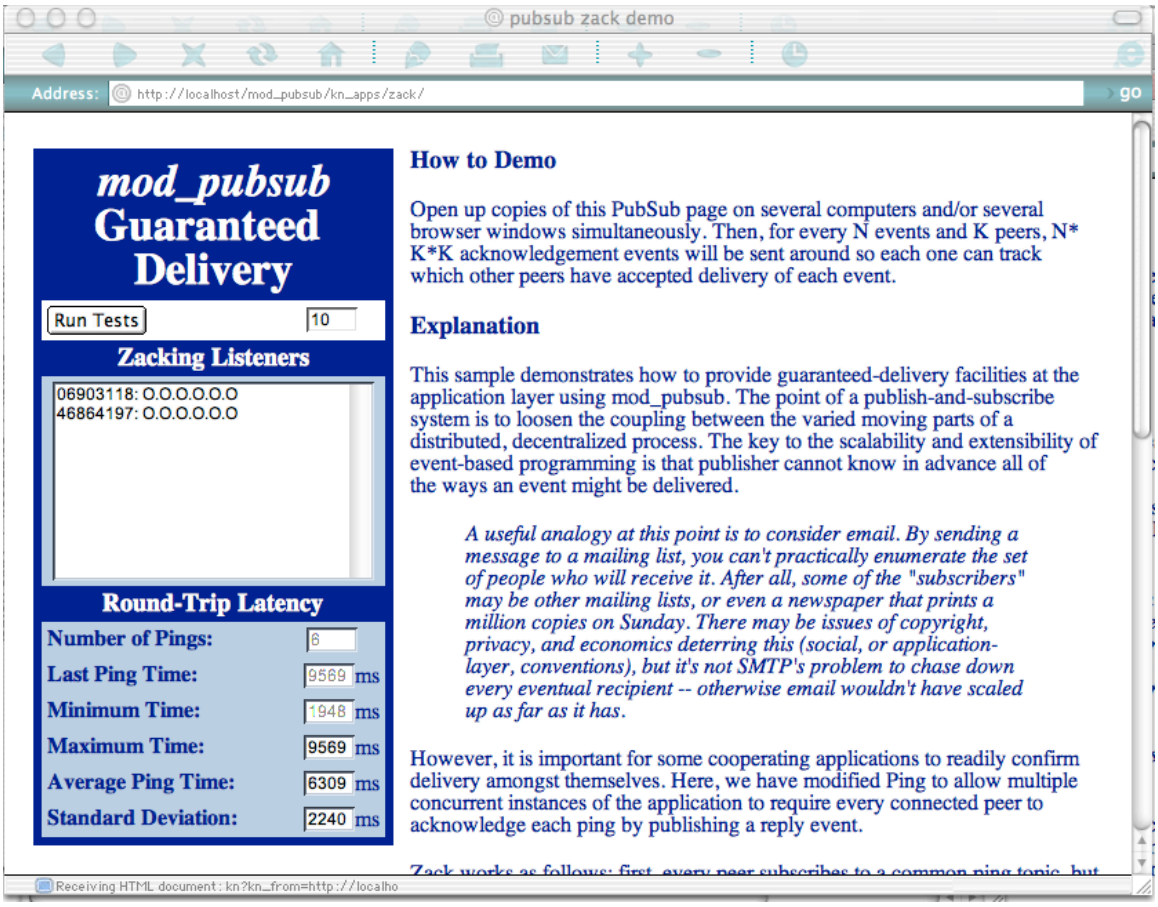

Figure 25: The ZACK acknowledgment-counting sample application.

Part of what ZACK demonstrates is how to extend an event-based system incrementally, since it is a voluntary additional protocol whose correspondents discover each other using a user-defined ZACKING flag on their route events in ./KN_ROUTES. Note that since it was prototypes as a browser-based application, it also depends on a single router to relay messages over tunnel connections; there really aren't *N* individual routers.

For our purposes, the goal is to share a queue that indicates what order lock requests were made. Thus, one of the *N* possible publishers uses ZACK to publish the timestamp of their request; and after writing the new value to the 'real' topic, just deletes that request event (without ZACK, since the LockReleaseHandler() from Program 19 doesn't rely on simultaneous agreement over the queue by all participants, just the new leader).

10.2.3.3 Prediction (ARREST+E)

The four types of ESTIMATOR connectors in REST+E become much more powerful in ARREST+E. STOREANDFORWARD queues can replay past events; SUMMARIZERs can coalesce or drop event notifications; PREDICTORs can generate speculative values for disconnected resources; and a TRUSTMANAGER must enforce local policies. Each of these can be implemented using MOD_PUBSUB.

STOREANDFORWARD. Because PUBSUB.CGI has access to a persistent store of past events, organized by topics, it was straightforward to modify it to deliver past events to new subscribers as well. Previously, many event-based application architects built hybrid systems, accessing databases for "historical" data and merging it with "live" data from an event bus. Not only does our approach unify these programming models — using an indefinite KN_EXPIRES, it can be used a database of sorts, as for buddy lists in KNOWBUDDY — it also lays the foundation for store-and-forward event relaying.

The key is that occasionally-connected components can re-establish subscriptions and request some number of recent notifications (DO_MAX_AGE), some past time interval (DO_MAX_N), or since a pre-arranged 'checkpoint' from a prior connection (DO_SINCE_CHECKPOINT). The next more sophisticated approach would be to synchronize topic histories by merging lists of events, as in cache validation or Usenet news.

Note that while MOD_PUBSUB includes adaptors for "push" notifications such as sending events as email messages, it does so by using external components such as FORM2SMTP.CGI (not to be confused with MAILFROM.PLX, an inbound sensor). In practice, the prevalence of firewalls and NATs on the public Internet today makes it essential to support on-demand event forwarding.

SUMMARIZER. We can identify at least three major types of summarization, each of which can be found in different portions of the MOD_PUBSUB project: compression, coalescing, and expiration. The first is a mechanical matter of negotiating the use of,

say, GZIP to compress all downstream traffic to a modern Web browser or other microserver. The second is an application-specific transformation that, in effect, derives a new, lower-frequency or lower-bandwidth event source from the original. Whether by discarding messages that don't meet some query (KN_CONTENT_FILTER), taking a derivative or integral of a signal (vote-tallying in VOTE2 or leader-change alerts in FLASHDEMO1 voting), or another application-specific rule for compensating transactions, the result ought to be a reply stream that falls within the limits of the subscriber's network connection. The third is another mechanical test, to re-evaluate notifications queued for delivery in a KN_JOURNAL topic once it is finally written to a socket (see §10.3.2.1).

PREDICTOR. Prediction functions are extremely application-specific. The only generic behavior we identified is inertia, the principle behind REST+E. That is the default behavior of all the microservers in MOD_PUBSUB; they do not trigger invalidation notifications when events expire. Application developers are free to enforce or ignore expiry deadlines as they see fit.

TRUSTMANAGER. The essential role of a TRUSTMANAGER is to ensure that every connection between components in the architecture is justified by a corresponding edge in the Web of Trust (see §8.3.14). Since MOD_PUBSUB was first and foremost a working testbed, rather than a theoretical exercise, almost all of our experiments relied on persistent tunnel connections. Without a mechanism for subscribers to accept inbound messages from anywhere else on the Internet, 'spam' became impossible by definition — the router was presumed to be a trusted third-party.

Where the role of a TRUSTMANAGER re-emerged was for managing permissions *within* individual topics. Rather than merely controlling access to the server as a whole, we needed to control who could publish or subscribe by pathname patterns. Hence our convention for putting shared data under /WHAT (such as /WHAT/AUTOREQUEST), but putting private data intended for particular users under /WHO/<USERID>/.

We used reflection and recursion to express fairly sophisticated access controls using a single basic privilege: publishing event notifications. Consider privacy rules for an instant messaging application. You might want to control who can join your buddy list (anyone); who can send you messages (only buddies); and who can enumerate your entire buddy list (only you). By analogy to file permissions in Unix, that means /BUDDYLIST must be UG+RW (0660), /BUDDYLIST/KN_ROUTES must be A+W (0622) and /BUDDYLIST/KN_ROUTES/KN_ROUTES must be U+RW (0600). This was easy to implement with Apache, by logging into the host and configuring .HTACCESS files.

Particularly dynamic filtering could also to be enforced at the edges. To launch a game such as TICTACTOE, CONNECT4, or REVERSI, the application needed to use a topic that only those two players could publish to, in strictly alternating order. We could rely on the router's clock to arbitrate who moved first, but rather than at-

tempting to provision a new topic on the server with those permissions on the fly, our applications merely blocked extraneous 'moves' by filtering on an event's KN_USERID header. All components still had to trust the router to label messages correctly, but could implement their own policies on that basis.

10.2.3.4 Decentralized Decisions (ARRESTED)

We already established that ASSESSOR components are entirely application-specific; the only generic support they require is the ability to subscribe to multiple routers concurrently. Because of a quirk in the browser's security model, our primary development platform made it impossible to connect to more than one router at a time, and none outside of a limited domain (suffix-matching rules). Only later, in the commercial development of the platform did we develop .NET, JMS, and Excel microservers that could manage multiple tunnels simultaneously. As for the JavaScript microserver, our experiments were still limited to Web-of-Trust topologies that allowed a single router to consolidate all traffic on behalf of a given subscriber — all data from external agencies had to be proxied through it.

10.3 EVALUATION

It may seem remarkable that a development effort that preceded the invention of our new architectural styles by several years still includes enough evidence to validate their feasibility. Nevertheless, MOD_PUBSUB could be faulted for not implementing the new elements we have introduced directly. Instead, our indirect arguments are summarized in Table 15.

To validate our claims for the practicality of our new architectural styles, we must go beyond the evidence of feasibility. We also need to evaluate whether our infrastructure unduly limits the scalability or the range of applications that can be developed with it.

Style	New Element	Implementation
REST	GLOBALCLOCK	Use NTP to measure 'seconds since UNIX epoch.'
REST+P	POLLINGCLIENT	META-REFRESH already exists in HTML browsers.
A+REST	NOTIFYING-ORIGINSERVER	KN_SERV experiment held connections open to stream multiple replies using Dynamic HTML.
R+REST	ROUTINGPROXY	KN_CONTENT_TRANSFORM relays the *output* of a proxy.
REST+D	MUTEXLOCK	Filesystem locks ensure durability, serialization.
REST+E	TCP	TCP/IP stacks handle retransmission and acks.
	CACHE	Browsers and proxies already return stale replies.
	ACCESSCONTROL	Web servers have many authentication schemes.
	CONTENT-NEGOTIATION	Retrieving a single event as an ordinary resource invokes existing ACCEPT- selection behavior.
ARREST	CENTRALIZED-EVENTROUTER	./KN_ROUTES "subdirectories" contain all subscriptions originating from parent topic; can be nested.
ARREST+D	FAIRMUTEXLOCK	End-to-end implementation of BAKERY for clients.
ARREST+E	STOREANDFORWARD	DO_MAX_AGE and DO_MAX_N flags to replay events.
	SUMMARIZER	VOTE_COUNTER and KN_CONTENT_FILTER derive new streams; tunneling can support 'in-flight' expiration
	PREDICTOR	N/A — Implementation is application-specific.
	TRUSTMANAGER	Per-topic access controls and KN_USERID filtering.
ARRESTED	ASSESSOR	N/A — Implementation is application-specific.

Table 15: Summary of our new architectural elements and their implementations.

10.3.1 Scalability

MOD_PUBSUB has successfully been operated for months at a time; with up to 50M events stored; with hundreds of connected clients, with hundreds of events published per second; with thousands of notifications per second; and total end-to-end notification latency of as little as 50 milliseconds — but never at the same time.

It is critical to isolate the figures of merit that can characterize a router's performance. We developed several tools specifically for this purpose, such as `PERFTOOL`. This helped us identify several optimizations in the construction of our routers, but these can be termed accidental, rather than essential challenges [37]. There are many variables one might measure: aggregate bandwidth, jitter, memory usage, but only three suggest themselves as independent variables: the number of connections terminated, events processed, and filtering/transformation overhead.

CONNECTIONS. The first axis is simply the number of concurrent persistent TCP connections to be maintained. In future years, this may become less interesting as operating systems mature and continue to tackle efficiency improvements such as accounting for TCP control block interdependence [219]. Nonetheless, there will always be some computational overhead for every connected user, active or not.

In the meantime, we also developed a simple elaboration of `PUBSUB.CGI` that increased the number of concurrent connections dramatically. Our `PUSH_MANAGER` experiment split the tunnel-maintenance function from the event-publication and routing functions. This accelerated performance by using a single Linux process to terminate all tunnel connections separately, rather than using CGI's process-per-socket execution model.

FREQUENCY. The second axis is the event rate. It could be refined into average and burst frequencies, and clearly could be overwhelmed by very large events. A secondary level of detail would be characterizing peak event *publication* rates, as distinct from event *notification* rates.

PROCESSING. That caveat is reflected, indirectly, in the third figure of merit. If we assume that our capacity is limited by CPU speed and raw bandwidth, then the only other major use of CPU time is *filtering*, which can be arbitrarily complex. Under this category, too, are session-long transformations such as SSL, `GZIP` compression, or delta-coding [159], for example.

10.3.1.1 Clustering

All of these factors could become bottlenecks for the performance of a single router, but the ultimate basis for our claim that these styles can be implemented without loss of scalability is that they do not limit the performance of a *cluster* of routers.

When discussing the semantics of topics and events, we often related them to the files on disk. This analogy helps explain how we envision tuning router clusters to provide arbitrary levels of reliability, availability, and capacity.

The simplest way to think about the problem is to represent a router as a single disk. Suppose a disk can read (subscribe) and write (publish) complete files as a

reliable transaction.[29] Then, with two disks, a mirroring strategy is to read and write to both disks. If either operation might fail at probability $p\%$, N-way mirroring reduces it to $p^N\%$. This increases reliability/availability; furthermore, if you assume you can read from either disk in parallel, then you can improve capacity, too. This is the standard argument for Redundant Arrays of Inexpensive Disks (RAID, [178]).

Up to 8-way mirroring of this kind is already available in KnowNow's LiveServer. It can also be emulated with MOD_PUBSUB by setting up mutual subscriptions to topics across a cluster (duplicate suppression ensures 'flood-fill' behavior). The only complication is the additional time lag for event propagation across the cluster.

Within a single data center, one can assume the bandwidth and error rate is much, much higher than the links to the outside world. Forwarding events across a WAN soon encounters precisely the sorts of problems that motivate our investigation of decentralization in the first place: excessive latency can force cluster members 'out of sync' and the myriad agencies Internet transactions cross en route can play havoc with security precautions.

Not surprisingly, our solution would be to view the "router" itself as an application in ARRESTED style. In this case, the current representation of an event source becomes a matter of opinion, and our goal has to be minimizing the risk of such disagreement. As we have discussed earlier, the key challenge becomes identifying which other router's resources should be considered "equivalent," a matter of trust management.

Simple mirroring would imply that *all* of the other routers are equally valid authorities. Clearly, this cannot hold in the face of limited computing power, storage, and bandwidth. The very reason we are proposing clustering is to divide the load, so that only a subset of other routers must keep track of each event source. A hash function would appear to be a simple solution: locally compute the set of other routers which share responsibility for a given event source.

Consider a simple rule: that topics with an even number of letters go to router A, and an odd number to B. This can be enforced either at the routers, by redirecting users to the other, or at the clients, by widely publicizing the even/odd rule and the addresses of A and B. This corresponds to "hashing" and "striping" whole[30] topics across several routers.

---

[29] If it does fail, we presume it fails permanently. "Backup recovery" or "log reconstruction" are useful engineering techniques to reduce p, but they do not affect our basic argument.

[30] It is straightforward to automatically break "big" topics into subtopics (along the lines of 'inodes' in the UNIX filesystem [221]). If there are too many events in one topic, break it and cascade it to another, new topic on the other machine. If, instead, there are few events, but too many subscribers, the same algorithm still applies — since *./*KN_ROUTES is itself just another topic.

However, every client of this router cluster would have to agree *in advance* on how many routers are in the cluster, their addresses, and the hashing algorithm. Otherwise, havoc ensues when the membership of the cluster changes.

A better solution lies in the mathematical insight behind many Content-Addressable Networks (CANs, [185]): a consistent hash function. Rather than outputting a single, fixed value for each input like an ordinary hash, a consistent hash generates several outputs, each with a declining probability of being correct as the size of the cluster changes.

Using such a function, even clients that disagree about the current membership of the cluster can all still cooperate to read and write data to *approximately* the right disk. If one client thinks there are 3 members, and another thinks there are 4, the consistent hash assures us that we both still would place $^2/_3$ of the data on the same disk, and that if not, with decreasing probability on the second or third disk.

The original application this was developed for was Web caching (later commercialized by Akamai [117]). The same approach applies to our event notification service because it is essentially another sort of Web cache. Publishers must choose which caches to populate with their new representations; subscribers must choose which ones to `WATCH`.

10.3.2 Applicability

We began our study of event-oriented systems by assembling a bibliography enumerating over one hundred coordination and collaboration systems that build on the notion of events [189, 190]. Not all of these systems are explicitly event-oriented, nor do they all employ "event notification services" per se, but taken together, they revealed an evolutionary family tree. It is illustrative to consider each of these application domains — messaging, presence, conferencing, simulation & graphics, and software integration — since they cover different design regimes. As shown in Table 16 (excerpted from [191]), their events differ in frequency, distribution, and content; and have different naming models, event transformation hooks, and security concerns.

	Messaging	Presence	Conferences	Simulation & Graphics	Software Integration
Event Frequency	Minutes up to days	Minutes	Seconds	Milliseconds	Milliseconds up to hours
Topology	1-Many (news), 1-Known (mail)	1-K (buddies)	1-1 (chat), 1-K (lecture), K-K (forum)	1-1 or small K-K groups	1-K or anonymous broadcasts
Content	Text to multimedia	Short test	Text to multimedia	Small and stateful updates	Machine-readable streams
Naming	Newsgroups, mailboxes	Users, groups	Users, handles, channels	Participants, simulated elements	Processes, hosts, tools
Transformations	Compression, batch delivery	Batch update, state timeouts	Rendering	Aggregation, filtering, dead-reckoning	Data type conversion
Security	Authentication, confidentiality	Privacy	Authentication, confidentiality	Closed system	Access controls

Table 16: Properties of five major categories of event-based applications.

The routers we have built can conceivably address most of the spectrum of requirements identified for these application areas. The two fundamental limitations to their applicability are that we did not design it for lossy or low-latency networks.

This affects videoconferencing applications, for example, because multimedia streams have been optimized to work around lost IP packets. All of our work has focused on TCP sockets, which sacrifice real-time performance to avoid errors.

It also affects fast-paced multiplayer games. Traversing the public Internet typically requires at least tens of milliseconds; that is the regime for which our routers were designed. Achieving much lower latency for highly interactive simulations requires alternative implementation techniques, ranging from high-performance adaptive object brokers (TAO, [170]) up to hard-real-time scheduling systems.

10.3.2.1 Journaling

An interesting example of "unintended consequences" crept into the initial proto type design and still has not been polished out. A feature we created called "journaling" prevents our current generation of routers from actually implementing one of the most important aspects of event notification: the ability to update or discard out-of-date events. This limits the applicability of `Mod_PubSub` for low-bandwidth subscribers.

Since we emphasized portability for the `PUBSUB.CGI` prototype, we chose to use the local filesystem to store its event cache. 'Tunnel' connections would periodically poll the disk to detect updated event notifications to transmit. Since the minimum timestamp resolution of the filesystem was 1 second, this limited *every* subscriber to 1Hz — even if the event was being updated much more often.

Since this was an artificial limit, not a fundamental network latency or bandwidth problem, we decided to insert a filter that generated a unique sequence number for each event notification destined for a tunnel. Thus, "new" events were now generated for > 1Hz sources.

This was clearly inappropriate when the read frequency was *lower* than the write frequency. We compounded the mistake by relying on this behavior to simplify the microservers. If two local components held identical subscriptions, a local micro-*router* would be able to dispatch a copy of a matching event to both; but having the router send two copies of the same event only requires demultiplexing.

This was just one of many lessons we learned from implementing new applications using `MOD_PUBSUB`. The next chapter describes our experiences building just one, an auction scenario to illustrate how our style's claims are induced in practice.

Chapter 11: Applications

While we have already implemented a large number of sample applications, including some key testing, development, and administration tools, in this chapter we will focus on a specific auction scenario to illustrate our development methodology and claims for each style. The AutoMarket auction (Figure 26) is an archetypal example because its key resource, the price, can shift from sole control (Figure 27), to shared control (Figure 28), and to decentralized control (Figure 29).

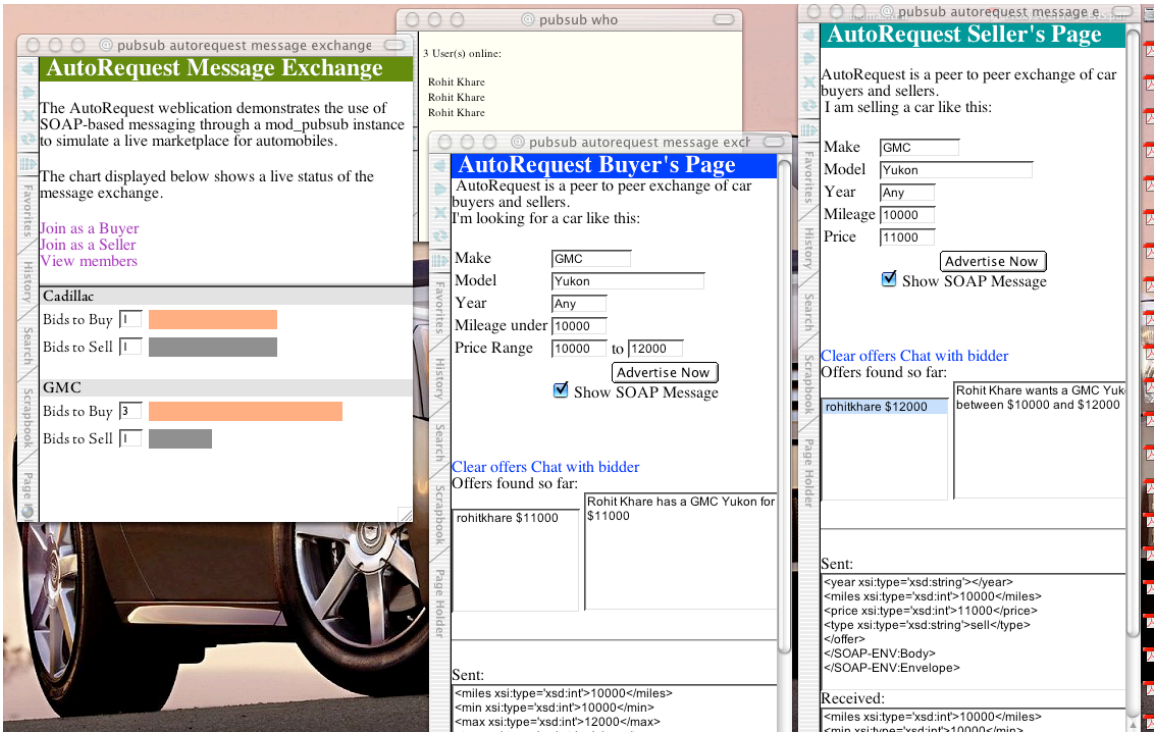

Figure 26: Screen shots of an example used-car marketplace.

In the following sections, we introduce different types of auction markets; propose a development methodology for ARRESTED-style applications; apply that methodology to the setting of a used-car marketplace; and recap our key observations from that experiment.

11.1 Auction Markets

Auction markets are an ideal setting for analyzing the consequences of decentralization. The equity market provides a brief illustration of the critical difference between centralized, distributed, estimated, and decentralized systems. Each represents a different type of process for discovering an equilibrium ('market-clearing') price.

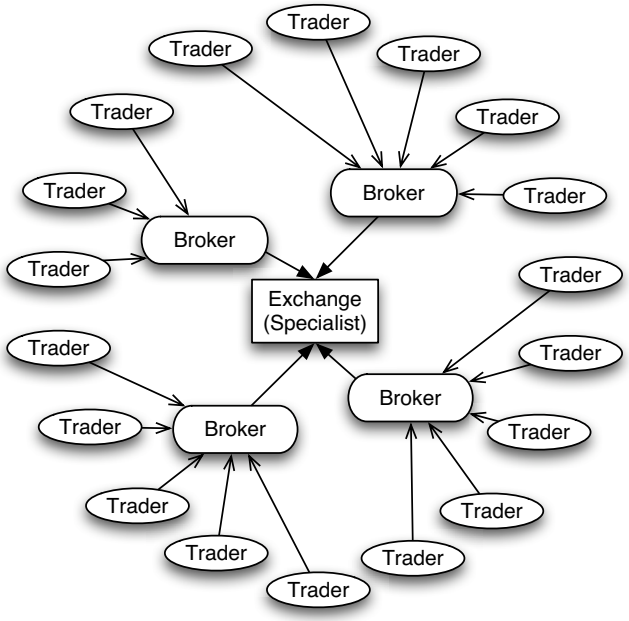

Figure 27: Illustration of a centralized (exchange) market.

The price of IBM stock is a centralized resource determined by a single 'specialist' at the New York Stock Exchange (NYSE). The price of Microsoft stock, by contrast, is determined by a consensus among a distributed set of dealers on the National Association of Securities Dealers Automated Quotation System (NASDAQ).

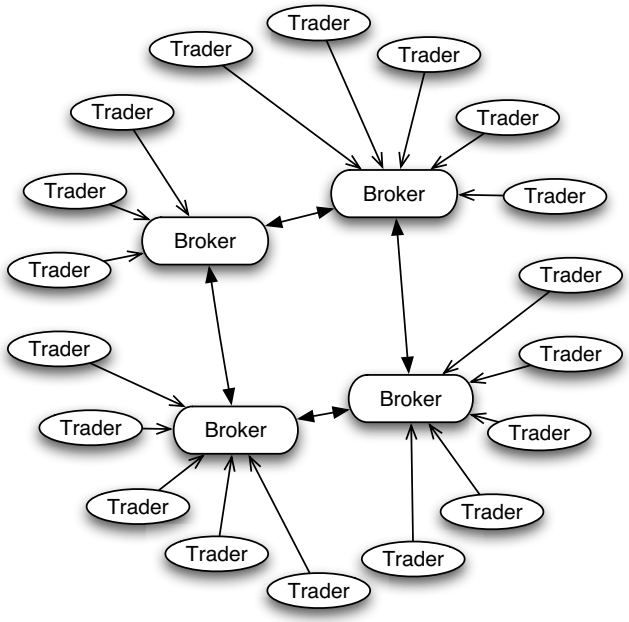

Figure 28: Illustration of a distributed (brokered) market.

Nevertheless, in either case there is only one 'true' price. Even if that price is changing as fast as 1Hz, a trader less than 500ms 'away' from the market can still buy or sell with confidence that the price has not changed by the time the trade is received.

Far more stocks, though, are traded on an 'over-the-counter' basis. The price of such thinly-traded companies cannot be reduced to a single official quote, because there is no such official authority. Every individual trade is a private matter, not an offer to the general public. At best, industry clearinghouses aim to publish weekly surveys of recent prices, hence the moniker 'pink sheets.'

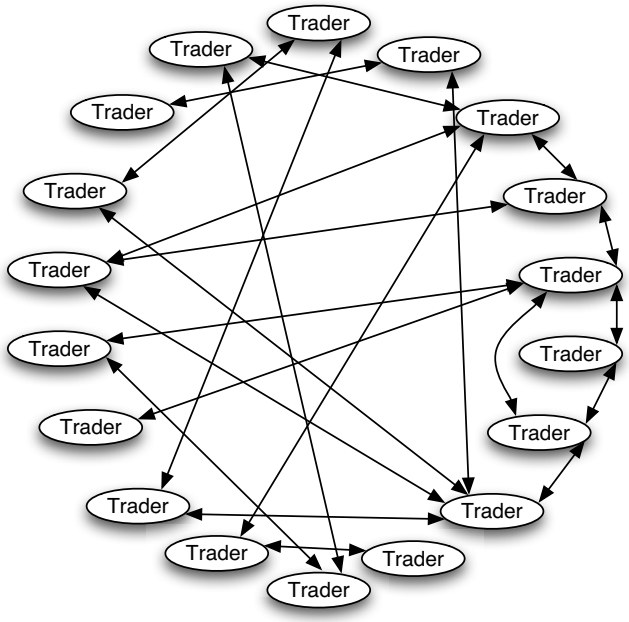

Figure 29: Illustration of a decentralized (over-the-counter) market.

A much larger-scale example is the trillions of dollars' worth of foreign currencies that are traded daily, largely without official currency conversion rates.[31] While there are many tradeoffs to such a decentralized market — the biggest players, known as "money center banks" can manipulate markets in ways considered illegal in stock trading — it is nearly completely fault-tolerant, proving more scalable than centralized markets both in theory and in practice. The foreign exchange market today is a continuous, 24-hour-a-day system. While equity markets struggle towards finalizing trades within three days ("T+3"), the Continuous Linking and Settlement Bank aims to settle within hours ("T+0") [4].

[31] Indeed, some financial research indicates that this uncertainty is the very reason the trading volume is so much larger than the actual demand for foreign currencies — all the 'excess' trades serve to signal information about the underlying market [105, 145].

11.2 METHODOLOGY

Since ARRESTED is an event-based architectural style, our development methodology for a decentralized application starts with the usual event-based modeling. One still has to identify the components, event sources, subscription qualifiers, message formats, and the like.

The second phase of our methodology specifically addresses the unique concerns raised by decentralization:

1. Identify the agencies.
2. Characterize the latencies.
3. Establish the web of trust.
4. Replace remote references with local estimates.
5. Expose the provenance of every event.

Consider how this methodology applies to an auction market. First, identify the locus of business logic and persistent state. In this case, this includes components representing traders, brokers, and an exchange. Next, identify the event model: suppose there is a topic for every commodity being auctioned; the notification format is to transmit the complete bid snapshot (rather than, say, a delta-coded increment/decrement 'ticker'); and that subscriptions can be based on the commodity name and triggered by numerical trends. Even this very-high level description is sufficient to sketch out a consensus-based application. It may perhaps even be sufficient for immediate prototyping using a parallel discrete event simulator [44].

Moving on to applying our additional steps for decentralization, we must elucidate the issues that arise once we consider the price to be a matter of opinion, rather than a matter of fact:

IDENTIFY AGENCIES. In any real marketplace, the interests of traders, brokers, and the exchange diverge significantly. None of the players are interested in complete transparency; there's far more money to be made in imperfect markets. That is the root of why so many different marketplace designs exist. Therefore, the very first step is acknowledging that every single component represents a separate agency —

sacrificing the conceit that a broker necessarily "represents" the best interests of its customers.[32]

Note that agency boundaries may also demarcate the remit of the architect's control of the application. A single organization may develop the software used to enact all of these agencies' roles, but the design must also be robust in the face of independent implementations. Part of the challenge of developing architectural styles for decentralization is coming up with abstract models of software written by others, software that may not even obey the architectural constraints one's own application relies on.

Characterize Latencies. The next step is to characterize the latencies, both of the networks the application will run on top of, and of the real-world phenomena that it is attempting to represent. Some players must be inside the now horizon to avoid taking on additional risks, and they might even cluster within the same city to minimize their communication latency. So the maximum latency for a high-end financial institution could easily be subsecond, necessitating the use of multiple telecommunications carriers and backup sites. By contrast, players attempting to trade stocks from PCs at home during volatile markets are well-advised to use limit orders instead. By the time a slow website loads and submits a form, the price could have shifted significantly.

That speaks to the second kind of latency we must characterize, namely that of the underlying phenomenon. Today, the interval between updates on the NYSE can range from fractions of a second to as long as three-day holidays (or even a week, after 9/11). However, that is in itself a reflection of the constraints imposed by our social and technological infrastructure, not the phenomenon itself.

Already, announcing news "after the closing bell" no longer permits the breathing room it once did: global stocks trade in global markets, around the clock. The valuation of something as complex as a company depends on so many inputs, which in turn vary at rates from seconds (online orders) to months (oil futures), that the end result could be said to be very high-frequency indeed. How high? Segments of the interbank foreign exchange market are already trading with five-second bid/ask exposures, worldwide. Stock prices could easily pass 1Hz as trading volumes continue to compound. Architects must take care to ensure that both kinds characteristic latencies are in equilibrium.

---

[32] We the Subscribers, Brokers for the Purchase and Sale of Public Stock, do hereby solemnly promise and pledge ourselves to each other, that we will not buy or sell from this day for any person whatsoever any kind of Public Stock, at a less rate than one quarter percent Commission on the Specie value of, and that we will give a preference to each other in our Negotiation.

— The "Buttonwood Agreement," May 1792, forerunner of the NYSE.

Web of Trust. The third step is establishing which resources controlled by other agencies ought to be considered equivalent — a web of trust between resources. In many consensus-based event-based architectural styles, the naming model alone determines equivalence — anything published by a duly authorized user to the /STOCK/IBM/BID/ topic must be a price quote to buy IBM stock. In a less certain world, though, there are many other criteria. Do you include the crazy fellow who's willing to bid twice as high as anyone else? Is the news feed entitled BUSINESS MACHINES, INTERNATIONAL relevant? Are brokers under indictment according to this other SEC.GOV website to be excluded from your view of the market? And why trust what the possibly-hacked SEC.GOV site says, anyway?...

Eliminate Remote References. This web of trust is critical for accomplishing the fourth step, eliminating all references to remote resources. Each reference must be replaced by two elements: a local proxy resource, and a formula for determining which other agencies' resources are considered trusted correspondents for that issue.

To isolate this complexity, our model guides architects to explicitly assessing the import of new information from foreign agencies before modifying private beliefs. It is an architect's choice between different prediction engines, compression engines, and other types of estimators. The result is a system that does not block unnecessarily while awaiting consensus with some remote resource.[33]

Track Provenance. The final recommendation of our methodology for decentralization is to track the provenance of every piece of data processed. In an era of profligate computing resources, event notification is an appropriate use of surplus bandwidth and audit trails are an appropriate use of surplus storage. Ideally, every datum displayed by a user interface ought to indicate its confidence interval, as well as the ability for a user to challenge the system to justify that estimate. This requires assessment functions that do not throw away provenance data, but rather, mine the data for patterns and other conclusions. After all, rumors from other traders that the crazy fellow is actually *paying* double might eventually tempt you into taking advantage of that offer as well — information from some players may affect your confidence in others, dynamically.

11.3 AutoMarket

To experiment with our new styles, we chose the domain of used-car sales. Following our methodology, the first phase of developing the AUTOMARKET application identified the components and events involved. BUYER and SELLER components publish BID events to separate /BUY and /SELL topics (rather than, say, BUY- or SELL-

[33] "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

— Leslie Lamport [137]

type BID notifications from a single /MARKET topic). Each user's BID events, in turn, must specify the make, model, mileage, and price; sellers must be specific, while buyers can specify ranges for acceptable mileage and price values.

We can now proceed to discuss the first three of the five steps specific to decentralization (the last two only arise later, in §11.3.3 and §11.3.4 respectively).

IDENTIFY AGENCIES. Each user is presumed to own his or her browser components, whether in the role of BUYER or SELLER. There must also be a single trusted-third-party operating the AUTOMARKET, who owns the web server running MOD_PUBSUB. Note that this third party is merely being trusted to run the event notification service honestly, not as a counterparty to any sale transactions.

CHARACTERIZE LATENCIES. In comparison to equity markets, used car sales are much slower-paced. It's a less liquid market, with nonuniform goods to boot. Considering the experience of current automobile auctions on the Internet, it's fair to say that while auctions may last for days, bids in the last few minutes may occur as frequently as 1 Hz. As for network latency, we are discounting the possibility of arbitrage by assuming every trader encounters the same distribution of latencies.

WEB OF TRUST. The only firm rule is that each trader must trust the agency operating the AUTOMARKET server. Any trader can choose to accept or discard bids from others, depending on their own preferences. Nonetheless, there is an implicit assurance that all traders are using the same syntax for their BIDS, including common vocabularies for makes and models.

In the next four subsections, we will describe how this basic design was adapted to enact centralized, distributed, estimated, and decentralized auctions, respectively.

11.3.1 Centralized Auction

In a centralized market, a single agency controls which goods are for sale and their prices, without becoming a buyer or a seller itself. An informal model is the role of newspaper's classified ads. If we can term this agent a DEALER, then we can implement this kind of marketplace by assigning the DEALER control of the Web server running the AUTOMARKET.

Our simple expedient for testing this was by writing BID events directly into files on the server's disk. MOD_PUBSUB includes this 'backdoor' by virtue of the fact that it uses the filesystem as a backing store; it uses SOAP for its structured data format [31], as shown in Program 21; and it scans the disk at 1 Hz for new files or modified files to appear. This allows us to experiment with event notification without relying on any remote access for publishing new events (which would require the distributed decision functions of ARREST+D).

Because we synthesized BID events with a minimum lease time of one day (+86,400 seconds in the units shown above), it was clear there was enough time for

BUYERS and SELLERS to learn of new offers while they were still valid, even if that made the centralized market quite low-frequency indeed.

Nevertheless, using this jury-rigged mechanism to publish and update BIDs, AUTOMARKET still exhibits the key properties we claim for ARREST-style applications:

```
userid: rohitkhare
soapaction: true
kn_id: rohitkhare_buy
kn_route_checkpoint: 1058752933_1076_82
kn_time_t: 1058752933
content-type: text/xml
displayname: Rohit Khare
kn_expires: 1058839333

<SOAP-ENV:Envelope xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
      <offer xsi:type='xsd:string'>
         <type xsi:type='xsd:string'>buy</type>
         <make xsi:type='xsd:string'>GMC</make>
         <model xsi:type='xsd:string'>Yukon</model>
         <year xsi:type='xsd:string'></year>
         <miles xsi:type='xsd:int'>10000</miles>
         <min xsi:type='xsd:int'>10000</min>
         <max xsi:type='xsd:int'>12000</max>
      </offer>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Program 21: Format of a BID event in AUTOMARKET.

SIMULTANEOUS AGREEMENT. Because users could WATCH the /BUY and /SELL topics, it was easy to see that all could stay synchronized even as new cars came on the market rapidly.

MULTILATERAL EXTENSIBILITY. A related aspect of the AUTOMARKET application is that traders could chat directly with each other by extracting their respective KN_USERIDs from BIDs. Either party could then choose to have the conversation abbreviated to take less space by using the KN_CONTENT_TRANSFORM facility to invoke PGRSPK.CGI. Another related experiment is to use the SOAP parsing and data-typing facilities shown in Figure 22 to extract and forward prices to a currency conversion service from XMethods Inc.'s online registry.

11.3.2 Distributed Auction

In a distributed market, *multiple* agencies control which goods are for sale and their prices. The first sense in which AUTOMARKET requires distributed control is that, in order to eliminate the role of an external DEALER, BUYERs and SELLERs have to be able to publish their own BIDs. This requires the AUTOMARKET server to delegate control over each user's offers to that user.

The second sense in which it calls for distributed control is that the price a SELLER is willing to accept becomes the best price from multiple BUYERs. That is, in an increasing-price ('English') auction, the current price of a good is the maximum of all BUYERs' BIDs.

2-WAY ACID SIMULTANEOUS AGREEMENT. The first sense calls for 2-way sharing, or delegated decision making (REST+D). A user must acquire a lock from the AUTOMARKET server to update his or her own BID events. This is implemented in MOD_PUBSUB by pushing the concern for ACID transactions down to the level of the filesystem. The host operating system's filesystem arbitrates mutually-exclusive control over the shared resource between the various concurrent executions of PUBSUB.CGI.

***N*-WAY ACID SIMULTANEOUS AGREEMENT.** The second sense — calculating the maximum price — calls for *N*-way sharing, or distributed decision-making (AR-REST+D). This experiment is as simple as embedding the same (shared) decision function in the BUYER and SELLER user interfaces (namely, displaying the maximum matching bid at all times).

Note that because there is no 'lost-update' problem — the only agency that can update a BID is its owner — there is no particular need to upgrade the system to use a FAIRMUTEXLOCK. Whereas, if the architect were to insist on distribution across *multiple* AUTOMARKET servers to increase reliability, that could still be accommodated by using the ZACK protocol for guaranteed delivery (see §10.2.3.2).

11.3.3 Estimated Auction

As we pass beyond AUTOMARKET's 'now horizon,' keeping the application responsive requires following the next step in our application development methodology. By replacing references to the DEALER's event sources with local estimates, ARREST+E can at least maintain the property of BASE Approximate Agreement. As discussed in §10.2.3.3, the primary application-specific component we are obliged to provide is a PREDICTOR.

ELIMINATE REMOTE REFERENCES. Since our example is not particularly high-frequency, the primary type of latency risk is disconnection rather than a few seconds' arbitrage. Per REST+E, the browser version of AUTOMARKET defaults to a policy of inertia by displaying any recent bid within the last 24 hours as its estimate.

To experiment with ARREST+E for SELLERs, we connected the same feed to KnowNow's Excel spreadsheet adaptor and used Microsoft® Excel's built-in time-series data processing functions to extrapolate current prices once disconnected. Given the specific behavior of an increasing-price auction, a logarithmic trend had a higher R^2 than a linear or polynomial curve fit. A similar risk-management approach applies for BUYERs: the use of range specifications in offer BIDs, which was merely a user-interface convenience until now, can now be cast as a limit-order.

11.3.4 Decentralized Auction

To complete the range of AUTOMARKET scenarios, we need to identify an abstract concept that multiple parties can legitimately disagree about. That is, an essential requirement for consensus-freedom, rather than the accidental consequences of network or process failures. We propose working with a characteristic of the market not already represented in a BID data structure: market segments. Using ARRESTED style to model an abstract category such as "trucks" obliges us to provide an ASSESSOR, as well as complete the fifth step in our development methodology.

TRACK PROVENANCE. With all of the SUV's and 'crossover' vehicles on the market, it's clear that the Federal regulatory definition of "trucks" encompasses far more than many users' notions of a pickup truck. However, since truck-ness is in the eye of the beholder, it isn't the place of SELLERs to classify their vehicles as cars or trucks; nor of the DEALER to enforce a single hierarchy such as /TRUCKS. Only the BUYER can say, and the application interface is obliged to justify why certain vehicles are included or excluded.

To experiment with ARRESTED, we used the 'command-line' parameter KN_TOPIC to redirect AUTOMARKET from using its default topic, /WHAT/AUTO-REQUEST/, to using an individual /WHO/<BUYER>/AUTO-REQUEST/ topic. Of course, sellers aren't aware of this switch, so they can't post offers directly to the BUYER's personal topic. Using INTROSPECT, a BUYER can proceeded to add routes connecting the public market to his or her view, while filtering certain models in or out of scope. As a simple example, adding a route with a KN_CONTENT_FILTER of "Escalade SUT" would only pass BIDs for the truck-like variant of the Cadillac Escalade (but not the regular SUV variety).

Formally, an ASSESSOR is responsible for two aspects of the Web of Trust: both establishing equivalence between others' names for the same concept, and the more common variety of authentication & authorization policy. The content-routing experiments focus on the former, rather than investigating security issues more thoroughly. To be sure, as long as network topology forces all traders to establish persistent TCP tunnels to a single router to circumvent firewalls and NATs, the Web of Trust between participants must also be restricted to star-topology trusted-third-

party relationships. This makes experiments in security trust management somewhat artificial until multiple routers are involved.[34]

11.4 Evaluation

Our new architectural styles and development methodology enabled us to transform an off-the-shelf auction application into a decentralized one. The Mod_PubSub project provided the infrastructure for a real-time event-based user interface, while we provided the application-specific prediction and assessment logic. This demonstrated that not only was it feasible to implement a generic infrastructure for our family of architectural styles, but that it was also practical to apply them in a realistic problem domain.

Auctions are a familiar application on the Web, but the usual REST-style experience leaves much to be desired. Users must poll for updated prices, while also deferring to the Web site owner's sole judgment of which buyers and sellers are reputable enough to trade with.

As our methodology suggested, the first step was developing an event-based model of an auction market. To complete this phase, we needed to identify the conflicting interests of BUYERs and SELLERs; ensure that the network latency was lower than the rate of price changes; and overlay our desired trust relationships on an underlying trusted-third-party network.

This enabled us to describe, first, a centralized market with a single advertiser of new offers; and then a distributed market with a shared decision function to determine prices. In either case, traders' browser-based user interfaces presumed consensus with remote resources managed by the router.

Shifting towards consensus-freedom, though, required us to provide two application-specific components beyond those in Mod_PubSub. The fourth step from our methodology required replacing references to the router with local estimates, in this case generated by a spreadsheet with logarithmic forecasting. The final step was shifting from monitoring individual vehicles to abstract categories like "trucks." The decentralized variant of AutoMarket tracked the provenance of such averages by creating personalized views for each trader that only filtered in qualifying BIDs.

In Table 17, we would like to recap our evidence that our styles deliver on our claims. As in the prior chapter, our original implementation of AutoMarket consid-

[34] While KnowNow's commercial product line includes microservers that can connect to multiple event routers (as a Windows DLL and in Java), the browser's security model restricted our early prototypes to only working within the same domains. So far, none of the open-source microservers in Mod_PubSub support connections to multiple routers in multiple domains.

erably predates the invention of our new architectural styles. As a result, some of our arguments are based on configurations for using it, rather than directly implementing new components.

	Claim	Experiment	Observation
ARREST	Simultaneous invocation of multiple services.	Operating the router like a 'classified ad' server by writing new BIDs directly to disk.	New information triggered updates of users' displays; could invoked per-user active proxies like abbreviation/conversion.
ARREST+D	Allow many clients to read *and* write to a shared resource reliably (ACID).	Operating the router as a passive relay of BID events controlled directly by each user.	Event notification enabled users to update all copies of their BIDs in all other components as soon as possible.
ARREST+E	BASE allows disconnected users to predict current prices.	Connecting the prices to an Excel spreadsheet to plot trends; using constraints on BIDs.	Fitting a logarithmic curve allows SELLERs to model increasing-price auctions; buyers can place 'limit orders' in advance.
ARRESTED	Consensus-freedom permits assessment of concepts.	Deriving a private topic from the public market data according to a trader's own filters.	Note how AUTOMARKET can provide synthetic estimates of the "truck" market, per each user's definition of a truck.

Table 17: Summary of observations from implementing AUTOMARKET in each style.

Chapter 12: Conclusions

In this dissertation, we proposed a new approach to decentralization, based on testable definitions and analysis of a formal model; new architectural styles that induce properties relevant for developing software for centralized, distributed, estimated, and decentralized systems; and developed infrastructure for, and applications in, each of our new styles. This chapter concludes by summarizing our contributions and proposals for future research.

12.1 Summary

Consensus-based architectural styles provide a simple and familiar programming model. When we refer to a register on a microprocessor, we expect to load its immediate value, not its past state. However, we cannot expect the same fidelity when memory and processors are decentralized across the Internet. In that case, we need new architectural styles that cope without consensus, permitting independent agents to hold multiple, simultaneously valid opinions about the value of a shared variable.

12.1.1 Problem Statement

We can decompose our research objective — designing styles of software architecture for decentralized systems — into a series of three sub-problems: What is the nature of "decentralization"? What architectural styles can induce properties that enable software engineers to cope with the challenges of decentralization? and Are such styles practical to implement and apply? These general questions, in turn, can be refined and restated as:

> What are the properties of centralized, distributed, estimated, and decentralized variables and resources?
>
> What new architectural elements and constraints can be added to REST to derive new architectural styles that support the use of centralized, estimated, distributed, and decentralized resources?
>
> Are there practical implementations for each new architectural element and constraint? Are these new styles usable for designing centralized, distributed, estimated, and decentralized applications?

12.1.2 Problem Analysis

Making a remote reference indistinguishable from a local one requires *consensus* over its value at each location. It becomes even more complex once that location can

be overwritten with a new value. In that case, we need to first establish consensus over the initial symbol and set an expiry deadline; then we can re-establish consensus over its new value (with a new lease). We termed this condition *simultaneous agreement,* since it connotes that it is not sufficient to establish that "the follower's value, once defined, is equal to the leader's value," but, rather, that "the follower's value, if defined, is equal to the leader's value *right now*."

This requirement induces a frequency limit for updating consensus-based resources. As a direct consequence of the well-known impossibility of consensus using asynchronous networks with faulty processes, no centralized (single-writer) resource can be modified more often than $1/d$ Hz, nor can any distributed (N-writer) resource be modified more often than $1/N \cdot d$ Hz, where d is the maximum latency of at least a partially synchronous network. The minimum interval between updates also lengthens in direct proportion to the number of possible process failures the system is designed to tolerate.

For any resource R changing at a maximum frequency F, then, we defined a boundary around the set of remote resources that can possibly establish simultaneous agreement with the value of R "right now." Our so-called *now horizon* denotes the subset of agents and resources that both trust R's owner and that are closer than $1/F$ seconds away from R.

Beyond the now horizon, there is uncertainty regarding both precision and accuracy when measuring R. Each local agent $A, B, C, \ldots$ will be forced to decentralize R into local proxy resources $R_A, R_B, R_C, \ldots$ and so on. Communication between local proxies is subject to network loss, delay, and congestion, all three of which increase message latency further. Depending on the degree of auto-correlation R exhibits, relying on older information can reduce the precision of local proxy estimates of R's the putatively current value. Furthermore, once local proxy resources are fully decentralized into an ensemble of independently controlled, local resources, such estimates also become less accurate. That is because there is no single 'true' value any more, since the *fact* of R has been replaced by a host of agency-specific *opinions* about R.

12.1.3 Insight

Our analysis identified clear reasons why practical large-scale systems cannot af ford the absolute certainty of simultaneous agreement. Instead, such systems need to accept the risk of manipulating out-of-date or untrusted data and manage that risk explicitly. That was our central insight for developing decentralized, or *consensus-free,* systems.

In particular, we organized our insights for risk management as a counterpoint to the well-established 'ACID' properties for maintaining simultaneous agreement in distributed systems. Our so-called 'BASE' properties identified the requirements for decentralized systems to rely solely on Best-effort network messaging; to Approxi-

mate the current value of remote resources; to be Self-centered in deciding whether to trust other agencies' representations; and Efficient in using network bandwidth.

12.1.4 Approach

In the second phase of our investigation, we proposed several new styles of software architecture that induce the properties necessary for manipulating centralized, distributed, estimated, and decentralized resources. We chose to start with REST, a network-based architectural style, and C2, an event-based architectural style.

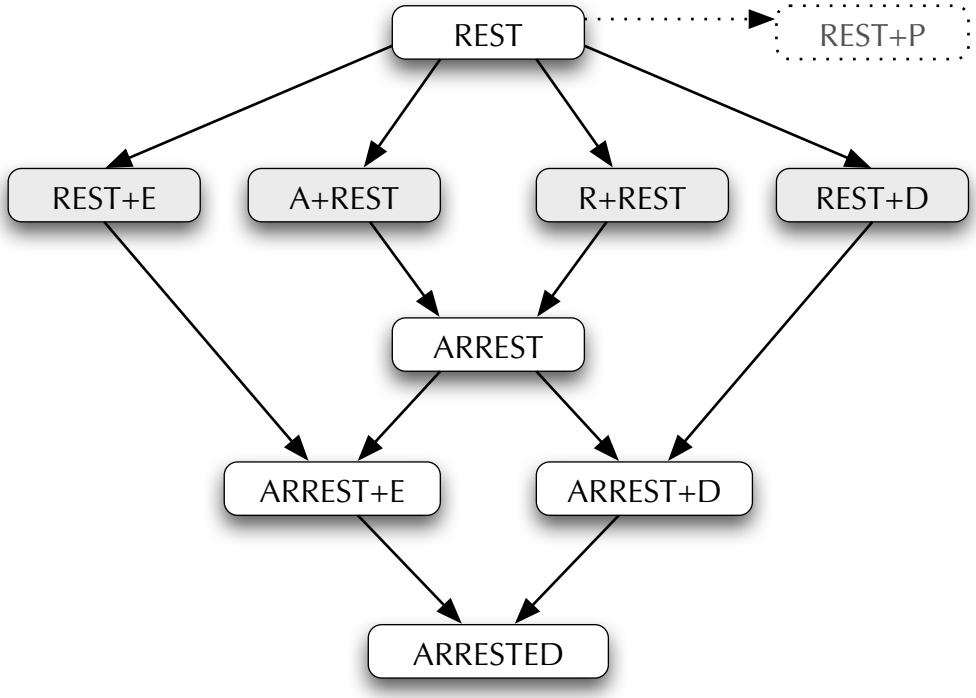

Figure 30: Diagram summarizing derivation of our four new architectural styles.

From REST, we derived several orthogonal features, each shown in the gray row of Figure 30. These features could be combined with each other to address the challenges of crossing the "now horizon" (Figure 31) and crossing agency boundaries (Figure 32).

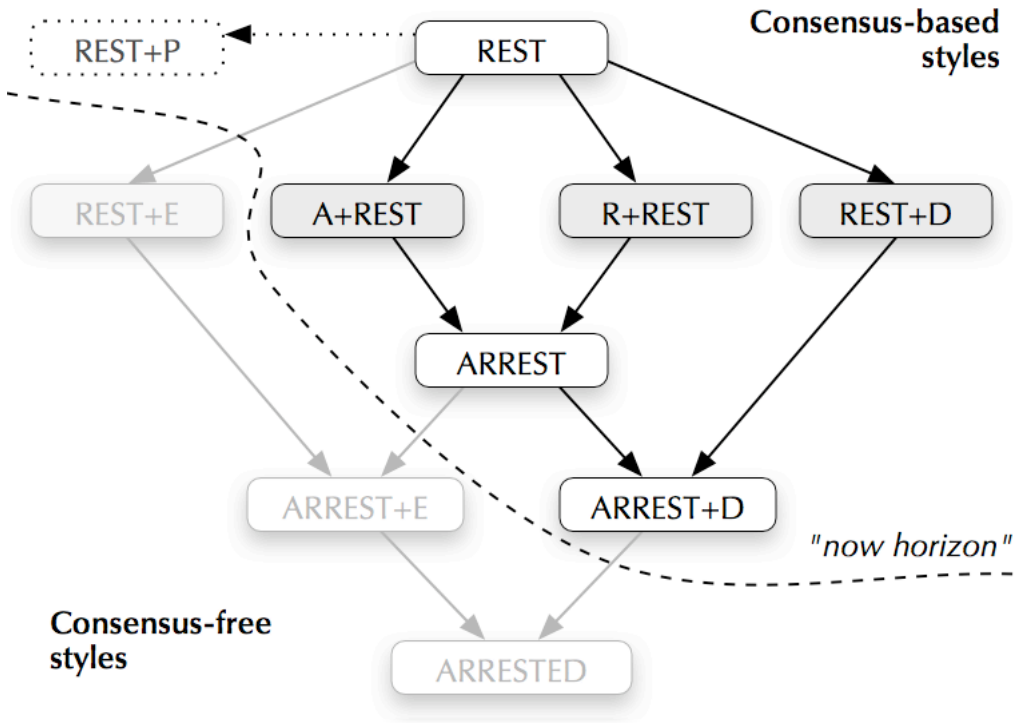

Figure 31: Consensus-based styles only work when the entire application is inside of the now horizon.

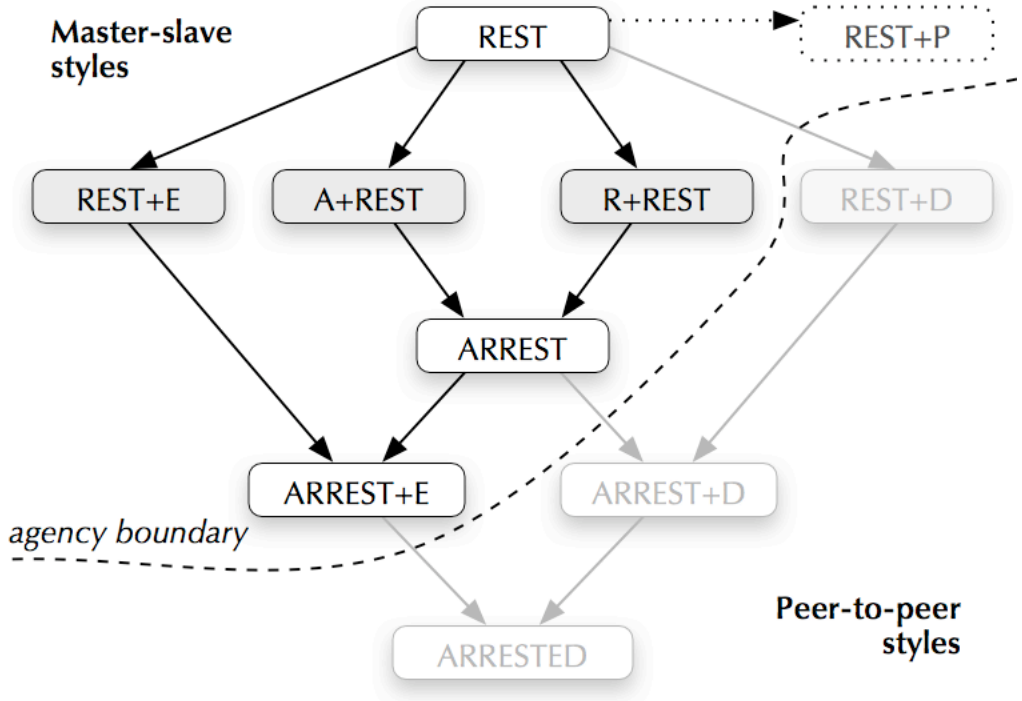

Figure 32: Master/slave styles only work for applications controlled by one agency.

12.1.4.1 New Centralized Styles

Unmodified, REST only enables consensus for write-once centralized resources. Even with synchronized expiration times and client-driven Polling (REST+P), its maximum update rate is three times slower than the theoretical limit. Thus, our first goal was to support mutable centralized resources. This requires simultaneous agreement over time, which Asynchronous REST (A+REST) enforced by replacing the RPC-like `GET` method with the event-based `WATCH` method. That enables a resource's origin server to broadcast notifications to its followers each time its representation(s) change.

We also want to generalize REST to permit more than one independent agency to extend an application. Its key restriction was that a linear proxy-chain model only allows hop-by-hop control of the entire path. More significantly, linear chaining also requires complete mutual trust, otherwise data could be modified by hostile upstream or downstream resources. Our approach is to pair both client and server connector types to simulate peer-to-peer connectivity, and then proceed to replace the *ad hoc* selection of proxy chains in REST with explicit control for message Routing (R+REST).

The combination of both techniques — routing event notifications across a graph of software components — yielded a new architectural style that enforced simultaneous invocation using publish/subscribe integration to synchronize local processing of centralized resources (ARREST).

	Goal	New Elements	New Constraints	Induced Property
REST	Refer to a centralized resource.	GLOBALCLOCK makes explicit how clients, servers, and caches are synchronized.	ORIGINSERVER must always specify a consistent expiry deadline if the resource is ever to be updated.	*Consensus:* Ensures that local resource proxies ***could*** agree with leader's value.
REST+P	Refer to a *mutable* centralized resource.	POLLINGCLIENT that reissues a new GET immediately upon each expiration.	Polling request rate must exceed $1/2d$ Hz.	*"Slow" Simultaneous Agreement:* Local proxies ***will*** agree with leader if its update rate is less than $1/3d$ Hz.
A+REST	Refer to a mutable centralized resource.	NOTIFYINGORIGINSERVER that can send multiple responses to a WATCH request.	Every resource update must lead to transmission of a new representation to all watchers.	*Simultaneous Agreement:* Ensures that local resource proxies ***will*** agree with leader's value, even if it is being updated at $1/d$ Hz.
R+REST	Compose services provided by multiple agencies.	ROUTINGPROXY Component that permits clients to control relaying.	Every representation transfer must be justified by a corresponding edge in the web of trust.	*Multilateral Extensibility:* Can compose trusted invocations without requiring mutual trust.
ARREST	Refer to the results of services that depend on centralized event sources.	CENTRALIZEDEVENTROUTER Component combines facilities to recast Resource/ Representation as an Event Source/ Event Notification model.	Notifications are relayed directly *through* services (proxies) to minimize latency.	*Simultaneous Invocation:* Ensure that services are invoked with the same inputs at the same time, every time.

Table 18: Summary of our centralized architectural styles.

12.1.4.2 New Distributed Styles

Having created a subscription mechanism for consensus-based systems, our second goal was to add a mechanism for other agents to publish updates. This requires distributing control of a resource amongst all the possible publishers. We add end-to-end Decision functions to share control across an ensemble of individual resources. To enforce ACID transaction properties, we mandated total serialization of updates by requiring simultaneous agreement over the decision function and all of its inputs (ARREST+D).

	Goal	New Elements	New Constraints	Induced Property
REST+D	Refer to a pairwise distributed read/write resource reliably.	`MutexLock` Component ensures only one client at a time has write access to the origin server.	Lock must be acquired before attempting write; then current state of the resource must be re-read before writing.	*ACID (Pairwise) Simultaneous Agreement:* Clients can modify centralized resources within $3d$ — but only in the absence of contention.
ARREST+D	Refer to an N-way distributed read/write resource reliably.	`FairMutexLock` Component to arbitrate shared locks with bounded bypass.	For a peer-to-peer solution, all N must cross-subscribe to each other's centralized ticket variables.	*Atomic, Isolated, Durable*: Ensure updates apply to all N resources reliably within $2d$ and at most $N{\cdot}d$.

Table 19: Summary of our distributed architectural styles.

12.1.4.3 New Estimated Styles

Ultimately, our aim is to keep entire applications functioning in the absence of consensus through *decentralization*: permitting independent agents to make their own decisions. Thus, our third goal was adding support for estimates that stand-in for remote resources that lie beyond the now horizon or agency boundaries.

This requires accommodating four intrinsic sources of uncertainty that arise when communicating with remote agencies: loss, congestion, delay, and disagreement. Their corresponding constraints are Best-effort data transfer, Efficient summarization of data to be sent, Approximate estimates of current values from data already received, and Self-centered trust management.

Our first exercise was to show how REST exhibits BASE properties once its standard components are deployed on an asynchronous, faulty network. Since REST was developed to explain the success of a robust, practical system, it is no surprise that it already includes features for coping without consensus (REST+E):

Best-effort representation transfers are pushed down to the presentation layer of the network using TCP's sliding window acknowledgement and retransmission protocols.

Approximate representations are returned by caches of several sorts: browser histories, caching proxies, and content distribution networks. Staleness is generally acceptable, even preferred in some cases.

Self-centered trust management is enforced by the use of server-based access controls, such as usernames and passwords.

Efficient representation formats are selected by client-driven content negotiation and dynamic content-transfer-encodings such as compression.

	Goal	New Elements	New Constraints	Induced Property
REST+E	Refer to a read-only centralized resource beyond its 'now horizon.'	`[TCP/IP]` `[Cache]` `[AccessControl]` `[Content-Negotiation]`	Inertia assumes that the most recent representation is still valid, until cache revalidation fails.	*Approximate agreement*. The local proxy should be in agreement *P%* of the time.
ARREST+E	Refer to a read/write resource *connected by a faulty network* beyond its 'now horizon.'	`StoreAndForward` Connector that adds end-to-end retransmission and acknowledgement policies.	End-to-end retransmission of notifications and acknowledgments.	*Best-Effort data transfer*: Cope with message loss.
		`Predictor` Connector for encapsulating Turing-complete prediction functions of past states.	Predict probable current state from past data (when possible).	*Approximate estimates*: Cope with message delay.
		`TrustManager` Connector that drops notifications from untrusted sources.	Ensure that all reachable endpoints are also trusted.	*Self-Centered trust management*: Cope with dynamic participation.
		`Summarizer` Connector to resample queued events at lower frequency.	Enforce bandwidth limits; Prohibit transmission of superceded data.	*Efficient data transfer*: Cope with network congestion.

Table 20: Summary of our estimated architectural styles.

Once we move to an event-based setting (ARREST+E), we can define more-sophisticated elements for each of the BASE properties because we have time-series data to work with. End-to-end Estimator functions manage private proxy resources to replace references to shared resources. We introduced estimators that mitigate each of the BASE challenges: store-and-forward retransmission of lost or delayed notifications, predicting future values from past information already received, discarding information from untrusted sources, and summarizing past data so as to send only the latest information. Such extensions to REST can increase *precision* of an estimate of a single remote resource (ARREST+E).

12.1.4.4 New Decentralized Style

Increasing accuracy, however, depends on assessing the opinions of several different agencies (ARRESTED). In this case, though, each agency can use a private decision function rather than a shared one; and the inputs can be estimates rather than certain values (thus avoiding waiting for simultaneous agreement with every other peer).

	Goal	New Elements	New Constraints	Induced Property
ARRESTED	Decentralize control of a shared resource across disjoint 'now horizons'	ASSESSOR Component that manages the risk of inter-agency disagreement over the 'true' value using a panel of opinions.	Eliminate reliable references to remote resources; only contingent estimates remain.	*Consensus-freedom:* must not presume feasibility of consensus at all.

Table 21: Summary of our decentralized architectural style.

12.1.5 Evaluation

With the derivation of these new styles in hand, the third phase of our investigation was to implement the infrastructure for our proposed new types of components and connectors; and to implement applications in each style that indeed exhibit the predicted properties. We discussed several interoperable open-source and commercial event routers, each of which successfully supported a wide range of implementation languages and communication protocols by judicious extension of existing REST infrastructure, such as standard Web clients, servers, proxies, and the HTTP/1.1 protocol.

Finally, we illustrated our claims for our new architectural styles using both a wide range of application samples from the MOD_PUBSUB project, as well as a coherent

series of auction applications that span the range of centralized, distributed, estimated, and decentralized marketplaces. Specifically, we adapted a used-car marketplace to support: centralized control of sale prices; distributed control in a best-price auction; estimated control in a GUI interface that continues to display a price range when disconnected from the network; and decentralized control of a generic concept such as "Truck prices," assessed from a series of individual vehicle auctions.

12.2 Contributions

Our contributions fall into three categories: our models, our styles, and our infrastructure.

First, we specified a formal model and definitions of the ill-defined terms centralized, distributed, estimated, and decentralized, as well as testable properties of each type of variable and resource:

> A *centralized* variable requires simultaneous agreement between a leader and its followers.
>
> A *distributed* variable is determined by applying a shared decision function over all participants' input variables.
>
> An *estimated* variable is in simultaneous agreement only a fraction of the time.
>
> A *decentralized* variable is determined by applying a private assessment function over other trusted participants' variables (or estimates of those variables).

Second, we derived an entire family of architectural styles from REST using several orthogonal features to induce several new properties, as shown in Table 22.

Style	**Induced Property**
A+REST	Simultaneous Agreement
R+REST	Multilateral Extensibility
ARREST	Simultaneous Invocation
ARREST+D	ACID Simultaneous Agreement
ARREST+E	BASE Approximate Agreement
ARRESTED	Consensus-Freedom

Table 22: Six new properties and the styles constructed to induce each.

Third, we developed open-source implementation for these styles, including multi-protocol application-layer event routers, management tools, and sample applications:

- The MOD_PUBSUB event routing module for Apache and other HTTP servers.
- The INTROSPECT application for monitoring and configuring an event router.
- The AUTOMARKET series of auction marketplace applications.

12.3 FUTURE WORK

Finally, let us return to the problems posed in the scenario of §1.1. What guidance can our ideas offer to an architect developing control software for a decentralized power web?

The challenge of writing an embedded control application for a household fuel cell may prove to be an instructive example. The goal, after all, is a worthy one:

> Development of a self-healing transmission and distribution system — capable of automatically anticipating and responding to disturbances while continually optimizing its own performance — will be critical for meeting the future electricity needs of an increasingly digital society. The benefits of a self-healing grid would include not only enhanced reliability, but also innovative customer services, real-time load management, reduced costs, and increased throughput on exiting lines via more-effective power-flow control. Standardized "plug and play" interfaces for both power and communications systems would allow distributed generation to proliferate. The self-healing grid would also increase grid security in response to the threat of terrorism. [232]

WHAT OUR MODELS CAN EXPLAIN. Designing software that reflects society requires representing the 'real world' — respecting the rights of citizens, communities, and corporations to make their own decisions independently.

First, our architectural styles prescribe a more detailed representation of a "fuel cell" than its physical properties alone. Any data structure representing a cell must describe it with respect to the interests of the agency that owns and operates it.

Second, our infrastructure also explains how a cell interacts with the network. By relying only on the BASE properties, the control software is in a better position to take advantage of any form of connectivity it has access to, from fiber and wireless Internet access to 'sneakernet' transfers by physical media — not limited to the real-time telemetry channels of today's grid control protocols (Supervisory, Control and Data Acquisition, SCADA [234]). Furthermore, it would be prepared from the ground-up for sustained network attacks, both in terms of degraded performance and partitioning, but also in terms of Byzantine deception.

Third, our design methodology prescribes a fundamental abstraction for the whole system: market equilibrium. In this case, the very purpose of a fuel cell is seen as participating in a local market with its neighbors. The basic coordination mechanism for all these independent agencies is to balance electricity production and consumption in real-time by setting market-clearing "prices" — whether or not that software abstraction is ever settled in terms of real-world currencies.

What our models can't explain. Architectural styles alone cannot solve the "essential" problems of controlling a fuel cell. Our job is to provide a framework (generic software infrastructure) for slotting in solutions to those parts of the problem. For example, we don't know what pattern of voltage fluctuations might indicate a dying cell, but ARRESTED at least can specify where to put the complex chemistry-simulation code that could make such predictions.

Benefits. How would decentralized fuel-cell-control software help society? Well, it *can* help us address some problems: how do emergency-services personnel shut it down — on whose authority? How might it react to active attacks that exacerbate price volatility?

On a more technical level, architectural models of applications that include annotations of agency boundaries and typical connector latencies hold promise for auto mating architectural critiques. Future work along these lines could even guide the physical layout of feasible configurations of processors, networks, and databases to meet the requirements of an application that requires a given update frequency.

Unsolved Problems. Of course, there are still myriad problems beyond the scope of our work at this point. Social challenges are among the most significant. By that, we mean that software is almost never a matter of programming to a mechanical, formal specification, but is an ongoing process of capturing and automating socially-constructed processes. This problem is simply not as closed-ended as steam-boiler-control.

A fuel cell exists not in isolation, but at the nexus of many different social processes. Its inputs depend on the price of oil — so should it issue commands to turn down the heat if CNN reports a missile strike in the Persian Gulf? Its power output is only a means, not an end in itself: should your fuel cell notice that your flight home has been canceled and turn off the power to the air conditioner tonight? Was that the only reason you needed it cooled, or are there pets? How sure would you want it to be that you hadn't caught another flight home? Should it check your cellphone records for confirmation of your whereabouts? What are the limits to this expanding model of integration into human planning?

This is a vaster canvas for application integration than anyone would imagine addressing today. Even scratching the surface of this vision depends on several new areas of future research: inference engines, assertion/metadata databases, and semantic web infrastructure. New techniques for decentralization could even assess event

notifications using historical trends, statistics, and even economic models — options, futures, Monte Carlo value-at-risk portfolio testing — in order to recover an approximate consensus even where it is no longer formally feasible.

Unsolvable (?) Problems. Finally, in conclusion, it is useful to consider a simple problem that we are even now powerless to solve: reasoning around fraud.

An archetypal example of the phenomenon we'd like to explain is the fake Emulex press release debacle [222], which temporarily wiped out $2B of investors' capitalization.

The first problem is how a fake press release gets to be circulated at all. When the latest news feeds from Reuters and AP *both* include a story about the press release from PRNewswire that Emulex was under investigation, one has a very strong estimate that, indeed, "PRNewswire.com says Emulex is under investigation."

However, this was still a highly inaccurate statement, even as it slipped by the wire services' fact checkers. The additional layer of concern for decentralized systems is to ask, "Do *I* believe that Emulex is *actually* under investigation?" To answer that, one might use software simulations that can juggle the possibility of both a true or a false outcome. It may well decide that the rewards for buying up the possibly-falsely-depressed stock exceed the likelihood that an actual scandal is occurring.

This seems much closer to the actual social model of trading. Software that purports to automate interaction with a decentralized stock market ought to reflect that. Perhaps, by putting a little colored indicator next to a news story reflecting how many organizations corroborate it, and how recently. Nevertheless, these are only technical means of testing whether an agency said something — not whether that *statement* is true.

After all, the credo of our postmodern age is that "truth is relative." If the urge for order and consistency is the wellspring for modern software architectures, perhaps our work can point the way towards *postmodern* software architecture.

References

[1] *Special Issue on Software Architecture* in *IEEE Transactions on Software Engineering*, 1995, *21* (4).

[2] *The Long, Dark Shadow of Herstatt* in *The Economist*, April 12, 2001. p. 70.

[3] *Accuracy is Addictive* in *Economist (Technology Quarterly)*, March 14, 2002.

[4] *Plumbing Revolution: Foreign-Exchange Settlement* in *The Economist*, November 16, 2002.

[5] Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. *Mach: A New Kernel Foundation for UNIX Development*, in *USENIX 1986 Summer Technical Conference*, (June 1986), pp. 93-112.

[6] Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M. and Chandra, T. D. *Matching Events in a Content-based Subscription System*, in *Principles of Distributed Computing 1999*, (1999).
http://www.research.ibm.com/gryphon/matching.pdf

[7] Aho, A. V., Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[8] Ahuja, S., Carriero, N. and Gelernter, D. *Linda and Friends* in *IEEE Computer*, 1986, *19* (8). pp. 26-34.

[9] Alateras, J., Mourikis, J. and Anderson, T. *OpenJMS*, ver. 0.7, Exolab, Inc., Melbourne, Australia, 2000-3. http://openjms.sourceforge.net/

[10] America Online. *AOLServer*, 1994. http://aolserver.com

[11] Anderson, J. *Lamport on Mutual Exclusion: 27 Years of Planting Seeds*, in *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, (August 2001), pp. 3-12.

[12] Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D. and Alberti, B. *RFC 1436: The Internet Gopher Protocol (a distributed document search and retrieval protocol)*. University of Minnesota / IETF, March 1993.

[13] April, C. A. *IBM rubs SOAP into MQ* in *Infoworld*, September 27, 2002.
http://www.infoworld.com/article/02/09/27/020930hnmqseries_1.html

[14] Association for Computing Machinery. *ACM Computing Classification System*. 1998. http://www.acm.org/class/1998/ccs98.html

[15] Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J. *Models and Issues in Data Stream Systems*, in *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, (June 2002). http://dbpubs.stanford.edu:8090/pub/2002-19

[16] Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O. and Spiteri, M. *Generic Support for Distributed Applications* in *IEEE Computer*, 2000, *33* (3). pp. 68-76.

[17] Beck, K. *Embracing Change with Extreme Programming* in *IEEE Computer*, 1999, *32* (10). pp. 70-77.

[18] Bell, D. E. and LaPadula, L. J. *Secure Computer Systems: Unified Exposition and Multics Interpretation, MTR-2997, Revision 1.* MITRE Corporation, Bedford, MA, March 1976.

[19] Bennett, C. H., Brassard, G., Crépeau, C., Jozsa, R., Peres, A. and Wootters, W. K. *Teleporting an Unknown Quantum State via Dual Classical and Einstein-Podolsky-Rosen Channels* in *Phys. Rev. Lett.*, 1993, *70*. pp. 1895-1899.

[20] Berners-Lee, T. *Web Architecture from 50,000 feet.* World Wide Web Consortium (W3C), 1999. `http://www.w3.org/DesignIssues/Architecture.html`

[21] Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F. and Secret, A. *The World-Wide Web* in *Communications of the ACM*, 1994, *37* (8). pp. 76-82.

[22] Berners-Lee, T., Fielding, R. and Masinter, L. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax and Semantics.* Internet Engineering Task Force, August 1998.

[23] Berners-Lee, T., Fischetti, M. and Dertouzos, M. L. *Weaving the Web : The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*, 1999. 226pp.

[24] Berners-Lee, T., Hendler, J. and Lassila, O. *The Semantic Web* in *Scientific American*, May, 2001. vol. *284* (5), pp. 34-43.

[25] Berners-Lee, T., Masinter, L. and McCahill, M. *RFC 1738: Uniform Resource Locators (URL).* Internet Engineering Task Force, December 1994.

[26] Birman, K. P. and Joseph, T. A. *Exploiting Virtual Synchrony in Distributed Systems*, in *Eleventh Symposium on Operating Systems Principles*, (Austin, Texas, 1987).

[27] Birrell, A. D. and Nelson, B. J. *Implementing Remote Procedure Calls* in *ACM Transactions on Computer Systems*, 1984, *2* (1). pp. 39-59.

[28] Blaze, M., Feigenbaum, J. and Lacy, J. *Decentralized Trust Management*, in *IEEE Symposium on Security and Privacy*, (DIMACS at Rutgers, New Jersey, May 1996), pp. 164-173.
`ftp://dimacs.rutgers.edu/pub/dimacs/TechnicalReports/TechReports/1996/96-17.ps.gz`

[29] Borbeley, A.-M. and Kreider, J. F. (eds.). *Distributed Generation: The Power Paradigm for the New Millenium* CRC Press, Boca Raton, Florida, 2001

[30] Bossaerts, P. L. and Øedegaard, B. A. *Lectures on Corporate Finance*. World Scientific Publishing Co., 2001. 248pp.

[31] Box, D., Enhnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S. and Winer, D. *Simple Object Access Protocol (SOAP) 1.1.* World Wide Web Consortium (W3C), 08 May 2000 2000.
http://www.w3.org/TR/SOAP/

[32] Braden, R. *STD-3/RFC-1122: Requirements for Internet Hosts - Communication Layers.* IETF, October 1989.

[33] Braunschvig, D., Garwin, R. L. and Marwell, J. C. *Space Diplomacy* in *Foreign Affairs*, 2003, *82* (4). pp. 156-164.
http://www.foreignaffairs.org/20030701faessay15411/david-braunschvig-richard-l-garwin-jeremy-c-marwell/space-diplomacy.html

[34] Breslau, L., Cao, P., Fan, L., Phillips, G. and Shenker, S. *Web Caching and Zipf-like Distributions: Evidence and Implications*, in *INFOCOM*, (1999), IEEE Press, pp. 126-134.

[35] Brewer, E. *Towards Robust Distributed Systems. (Invited Talk)*, in *Principles of Distributed Computing*, (Portland, Oregon, July 2000).

[36] Brewer, E. *Invariant Boundaries (Invited Keynote)*, in *9th International Workshop on High Performance Transaction Systems (HPTS)* (Pacific Grove, CA, October 14-17 2001).
http://research.microsoft.com/~jamesrh/hpts2001/presentations/hpts2001-brewer.ppt

[37] Brooks, F. P. *The Mythical Man-Month: Essays on Software Engineering* 2 ed. Addison-Wesley, 1995. 336pp.

[38] Brown, N. and Kindel, C. *Distributed Component Object Model Protocol – DCOM/1.0 (expired).* Microsoft, January 1998 1998.
http://www.alternic.org/drafts/drafts-b-c/draft-brown-dcom-v1-spec-03.html

[39] Burns, J., Paul Jackson, Lynch, N. A., Fischer, M. J. and Petersen, G. L. *Data Requirements for Implementation of N-process Mutual Exclusion Using a Single Shared Variable* in *Journal of the ACM*, 1982, *29* (1). pp. 183-205.

[40] California Public Utilities Commission. *Rule 21 on Distributed Generation (R99-10-025).* October 25 2000.
http://www.cpuc.ca.gov/static/industry/electric/distributed+generation/index.htm

[41] Carzaniga, A. *Architectures for an Event Notification Service Scalable to Wide-Area Networks* (Ph.D. Thesis), Politecnico di Milano, Computer Science, Milan, Italy, 1998. http://www.cs.colorado.edu/~carzanig/papers/index.html.

[42] Carzaniga, A., Rosenblum, D. S. and Wolf, A. L. *Design and Evaluation of a Wide-Area Event Notification Service* in *ACM Transactions on Computer Systems*, 2001, *9* (3). pp. 332-383.

[43] Cerf, V. G. and Kahn, R. E. *A Protocol for Packet Network Interconnection* in *IEEE Transactions on Communications*, 1974 (COM-22). pp. 637-648.

[44] Chandy, K. M. and Misra, J. *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs* in *IEEE Transactions on Software Engineering*, 1979, *5* (5). pp. 440-452.

[45] Chandy, K. M. and Misra, J. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988. 512pp.

[46] Chandy, K. M. and Taylor, S. *Introduction to Parallel Programming*. Jones & Bartlett, 1991. 228pp.

[47] Chon, K. *Information Processing in Electricity Distribution Systems*, in *Proceedings of the Annual Conference (Volume 2)*, (1978), ACM Press, pp. 979-984.

[48] Chung, P. E., Huang, Y., Yajnik, S., Liang, D., Shih, J. C., Wang, C.-Y. and Wang, Y.-M. *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer* in *C++ Report*, January, 1998.
http://research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html

[49] Clark, D. D. and Blumenthal, M. S. *Rethinking the Design of the Internet: The End-to-End Arguments vs. the Brave New World.* 2000.
http://ebusiness.mit.edu/research/TPRC-Clark-Blumenthal.pdf

[50] Clemm, G., Amsden, J., Ellison, T., Kaler, C. and Whitehead, E. J. *RFC 3253: Versioning Extensions to WebDAV*. IETF, March 2002.

[51] Collins-Sussman, B. *The Subversion Project: Buiding a Better CVS* in *Linux Journal*, 2002. (94). http://www.linuxjournal.com/article.php?sid=4768

[52] Committee on Payment and Settlement Systems of the G-10. *Settlement Risk in Foreign Exchange Transactions.* Bank for International Settlements, Basel, Switzerland, March 1996. http://newrisk.ifci.ch/137230.htm

[53] Crispin, M. *RFC 3501: Internet Message Access Protocol, Version 4rev1 (IMAP4rev1).* Internet Engineering Task Force, March 2003.

[54] Crocker, D. H. *RFC 822: Standard for the Format of ARPA Internet Text Messages.* IETF, August 1982.

[55] Curtiss, P. *Control of Distributed Electrical Generation Systems* in *ASHRAE Transactions*, 2000, *106* (1).

[56] Day, M. *Presence and Instant Messaging via HTTP/1.1: A Coordination Perspective*, in *Third International Conference on Coordination Models and Languages (COORDINATION '99)*, (Amsterdam, The Netherlands, April 1999).

[57] Delaney, J. R. *High-Speed Dial-Up: Does It Work?* in *PC Magazine*, June 17, 2003. http://www.pcmag.com/article2/0,4149,1090228,00.asp

[58] Denning, D. E. *A Lattice Model of Secure Information Flow* in *Communications of the ACM*, 1976, *19* (5). pp. 236-243.

[59] Dickerson, C. *The Battle for Decentralization* in *Infoworld*, May 2, 2003. http://www.infoworld.com/article/03/05/02/18OPconnection_1.html

[60] Dusseault, L. *WebDAV: Next-Generation Collaborative Web Authoring*. Prentice-Hall, 2003. 544pp.

[61] Ellis, C. A. and Gibbs, S. J. *Concurrency Control in Groupware Systems*, in *1989 ACM SIGMOD International Conference on Management of Data*, (Portland, Oregon, 1989), ACM Press, pp. 399-407.

[62] Evans, D. and Schmalensee, R. *Paying with Plastic: The Digital Revolution in Buying and Borrowing*. MIT Press, Cambridge, MA, 2000. 392pp.

[63] Feller, W. *An Introduction to Probability Theory and Its Applications*, vol. 1 of 2 3rd ed. John Wiley & Sons, 1968. 528pp.

[64] Fetzer, C. and Cristian, F. *An Optimal Internal Clock Synchronization Algorithm*, in *Compass '95: 10th Annual Conference on Computer Assurance*, (Gaithersburg, Maryland, 1995), National Institute of Standards and Technology, pp. 187-196. http://www.research.att.com/~christof/papers/preprint-COMPASS1995.pdf

[65] Feynman, R. P. *The Character of Physical Law*. MIT Press, Cambridge, MA, 1965. 173pp.

[66] Feynman, R. P., Allen, R. W. and Hey, A. J. G. (eds.). *Feynman Lectures on Computation* Perseus, 2000. 320pp.

[67] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures* (PhD Thesis), University of California, Irvine, Information and Computer Science, Irvine, CA, 2000.

[68] Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.* Internet Engineering Task Force, June 1999.

[69] Fielding, R., Whitehead, E. J., Anderson, K., Oreizy, P., Bolcer, G. A. and Taylor, R. N. *Web-based Development of Complex Information Products* in *Communications of the ACM*, 1998, *41* (8). pp. 84-92.

[70] Fielding, R. T. and Taylor, R. N. *Principled Design of the Modern Web Architecture* in *ACM Transactions on Internet Technology (TOIT)*, 2002, *2* (2). pp. 115-150.

[71] Fischer, M. J., Lynch, N. A. and Paterson, M. S. *Impossibility of Distributed Consensus with One Faulty Process* in *Journal of the ACM*, 1985, *32* (2). pp. 374-382.

[72] Flanagan, D. *JavaScript: The Definitive Guide*. O'Reilly & Associates, 2001. 900pp.

[73] Foster, I., Olson, R. and Tuecke, S. *Productive Parallel Programming: The PCN Approach* in *Journal of Scientific Programming*, 1992, *1* (1). pp. 51-66.

[74] Fox, A. and Brewer, E. *Harvest, Yield, and Scalable Tolerant Systems*, in *Proceedings HotOS-VII*, (1999).
http://swig.stanford.edu/pub/publications/harvest_yield.pdf

[75] Fox, A., Goldberg, I., Gribble, S. D., Lee, D. C., Polito, A. and Brewer, E. A. *Experience With Top Gun Wingman: A Proxy -Based Graphical Web Browser for the 3Com PalmPilot*, in *Proceedings of Middleware'98*, (Lake District, England, September 1998).

[76] Frank, M. P. *Physical Limits of Computing* in *IEEE Computing in Science & Engineering*, 2002 (May/June).

[77] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A. and Stewart, L. *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication.* IETF, June 1999.

[78] Freed, N. and Borenstein, N. *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* IETF, November 1996.

[79] Freed, N. and Borenstein, N. *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.* IETF, November 1996.

[80] Frystyk, H., Connolly, D., Khare, R. and Prud'hommeaux, E. *PEP: An Extension Mechanism for HTTP.* W3C Technical Report, November 1997.
http://www.w3.org/TR/WD-http-pep.html

[81] Frystyk, H., Leach, P. and Lawrence, S. *RFC 2774: An HTTP Extension Framework (Experimental).* IETF, February 2000.

[82] Frystyk, H. and Thatte, S. *Web Services Routing Protocol.* Microsoft, October 2001. http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-routing.asp

[83] Fuchs, M. *Dreme: for Life in the Net* (PhD Thesis), New York University, 1995.

[84] Fudenberg, D. and Tirole, J. *Game Theory*. MIT Press, 1991.

[85] Gall, H., Jazayeri, M., Klösch, R. and Trausmuth, G. *The Architectural Style of Component Programming*, in *Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, (Washington, DC, August 1997), IEEE Computer Society, pp. 18-25.

[86] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[87] Garfinkel, S. L. and Mahoney, M. K. *NeXTStep Programming: Step One: Object-Oriented Applications*. Springer Verlag, 1993. 629pp.

[88] Garlan, D. *Research Directions in Software Architecture* in *ACM Computing Surveys (CSUR)*, 1995, *27* (2).

[89] Garlan, D. *What Is Style?*, in *First International Workshop on Software Architecture*, (April 1995).

[90] Garlan, D. and Notkin, D. *Formalizing Design Spaces: Implicit Invocation Mechanisms*, in *VDM '91: 4th International Symposium of VDM Europe on Formal Software Development Methods*, (Noordwijkerhout, The Netherlands, 21-25 October 1991), Springer-Verlag, pp. 31-44.

[91] Garlan, D. and Shaw, M. *An Introduction to Software Architecture* in Ambriola, V. and Tortora, G. eds. *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, Singapore, 1993, pp. 1-39.

[92] Garlan, D. and Wang, Z. *A Case Study in Software Architecture Interchange*, in *Coordination'99*, (1999), Springer Verlag.

[93] Gelernter, D. *Mirror Worlds: Or the Day Software Puts the Universe in a Shoebox: How It Will Happen and What It Will Mean*. Oxford University Press, 1991. 237pp.

[94] George, J. F. and King, J. L. *Examining the Computing and Centralization Debate* in *Communications of the ACM*, 1991, *34* (7). pp. 62-72.

[95] Giesen, J., Wattenhofer, R. and Zollinger, A. *Towards a Theory of Peer-to-Peer Computability*, in *Proceedings of the 9th International Colloquium on Structural Information and Communication (SIROCCO)*, (Andros, Greece, June 2002).
`http://distcomp.ethz.ch/projects/p2p.html`

[96] Gilbert, S. and Lynch, N. A. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services* in *SIGACT News*, 2002, *33* (2).
`http://theory.lcs.mit.edu/tds/papers/Gilbert/Brewer6.ps`

[97] Gillmor, D. *Personal P-to-P Has No Peer. Mercury News*, San Jose, 16 February 2001, p. 1C.

[98] Gorlick, M. *The Role of Satellite Services in Internet Event Notification*, in *Workshop on Internet Scale Event Notification (WISEN'98)*, (Irvine, CA, July 13-14 1998).
`http://www.ics.uci.edu/~irus/wisen/wisen98/presentations/Gorlick/WISEN.2up.pdf`

[99] Gray, J. *Computer Technology Forecast for Virtual Observatories.* Microsoft Research, Redmond, WA, September 2000. 11pp.
`ftp://ftp.research.microsoft.com/pub/tr/tr-2000-102.pdf`

[100] Gray, J. (ed.), *Benchmark Handbook for Database and Transaction Processing Systems* Morgan-Kaufmann, 1993. 592pp.
http://www.benchmarkresources.com/handbook/tpca-4.html

[101] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, 1992. 1070pp.

[102] Guelich, S., Gundavaram, S. and Birznieks, G. *CGI Programming with Perl* 2nd ed. O'Reilly, 2000.

[103] Haber, S. and Stornetta, W. S. *How to Time-Stamp a Digital Document* in *Lecture Notes in Computer Science*, 1991, *537*. p. 437. http://www.surety.com/

[104] Halstead, R. *Multilisp: A Language for Concurrent Symbolic Computation* in *ACM Transactions on Programming Languages and Systems*, 1985, *7* (4). pp. 501-538.

[105] Hau, H., Killeen, W. and Moore, M. *How has the euro changed the foreign exchange market?* in *Economic Policy*, 2002, *17* (34). pp. 149-191.
http://faculty.insead.fr/hau/Research/Euro%20and%20FX.pdf

[106] Herlihy, M. *A Methodology for Implementing Highly Concurrent Data Objects* in *ACM Transactions on Programmmg Languages and Systems*, 1993, *15* (5). pp. 745-770.

[107] Hoare, C. A. R. *Communicating Sequential Processes* in *Communications of the ACM*, 1978, *21* (8). pp. 666-677.

[108] Hoff, T. E., Wenger, H. J., Herig, C. and Shaw Jr., R. W. *Distributed Generations and Micro-Grids*, in *18th Annual North American Conference of the US Association for Energy Economics*, (1997).

[109] Horton, M. and Adams, R. *RFC 1036: Standard for Interchange of USENET Messages.* IETF, December 1987. http://www.faqs.org/faqs/usenet/cancel-faq

[110] International Organization for Standardization. *Information technology – Open Systems Interconnection – Remote Procedure Call (RPC).* ISO/IEC 11578:1996 [Defines UUIDs], Geneva, Switzerland, 1996.

[111] JaJa, J. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. 576pp.

[112] Javasoft. *Distributed Events Specification*, 1998.
http://www.javasoft.com/products/javaspaces/specs/ev.pdf

[113] JavaSoft. *Java Message Service API.* Sun Microsystems, Inc., Palo Alto, CA, August 27 2001. http://java.sun.com/products/jms/docs.html

[114] Jefferson, D. R. *Virtual Time* in *ACM Transactions on Programming Languages and Systems*, 1985, *7* (3). pp. 404-425.

[115] Kahle, B. *WAIS: A Foundation for Network Publishing*. Addison-Wesley, 1996.

[116] Kammer, P., Bolcer, G. A., Taylor, R. N. and Bergman, M. *Techniques for Supporting Dynamic and Adaptive Workflow* in *Computer Supported Cooperative Work (CSCW)*, 2000, *9* (3/4). pp. 269-292.

[117] Karger, D. R., Lehman, E., Leighton, F. T., Levine, M. S., Lewin, D. and Panigrahy, R. *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, in *ACM Symposium on Theory of Computing*, (May 1997), ACM Press, pp. 654-663.

[118] Khare, R. *Building a Perfect Beast: Dreams of a Grand Unified Protocol* in *IEEE Internet Computing*, 1999. pp. 89-93.

[119] Khare, R. *Message-Oriented Middleware and the Software Engineer* (MS Thesis), University of California, Irvine, Information and Computer Science, 1999. http://www.4k-associates.com/4K-Associates/moma.html.

[120] Khare, R. *What's in a Name? Trust. (Internet-Scale Namespaces, Part II)* in *IEEE Internet Computing*, 1999, *3* (6). pp. 80-84.

[121] Khare, R. *Who Killed Gopher? An Extensible Murder Mystery* in *IEEE Internet Computing*, Jan/Feb, 1999. vol. *3* (1), pp. 81-84.
http://www.ics.uci.edu/~rohit/IEEE-L7-http-gopher.html

[122] Khare, R. *The Two-Way Web: An Interoperable Foundation for P2P*, in *O'Reilly Peer-to-Peer Conference*, (San Francisco, CA, 14 February 2001).
http://conferences.oreillynet.com/cs/p2p2001/view/e_sess/1163

[123] Khare, R. *SOAP Routing: The Missing Link*, in *O'Reilly Emerging Technology Conference*, (Santa Clara, CA, 2002).
http://conferences.oreillynet.com/presentations/et2002/khare_rohit.ppt

[124] Khare, R. and Rifkin, A. *Composing Active Proxies to Extend the Web*, in *OMG-DARPA-MCC Workshop on Compositional Software Architectures*, (Monterey, CA, January 1998), ACM SIGSOFT Software Engineering Notes 23(3), pp.44-63.

[125] Khare, R. and Rifkin, A. *Scenarios for an Internet-Scale Event Notification Service (ISENS)*. IETF draft (expired), 13 August 1998.
http://www.ics.uci.edu/~rohit/draft-khare-notification-00.txt

[126] Khare, R., Rifkin, A., Sitaker, K. and Sittler, B. *mod_pubsub: an open-source event router for Apache*, 2002. http://mod-pubsub.sourceforge.net/

[127] King, J. L. *Centralized vs. Decentralized Computing: Organizational Considerations and Management Options* in *ACM Computing Surveys*, 1983, *15* (4). pp. 320-349.

[128] Kohnfelder, L. M. *Towards a Practical Public-Key Cryptosystem* (S.M. Thesis), MIT, Laboratory for Computer Science, Cambridge, MA, 1978.

[129] Krasner, G. E. and Pope, S. T. *A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80* in *Journal of Object Oriented Programming*, 1988, *1* (3). p. 26–49.

[130] Krishnamurthy, B. and Arlitt, M. *PRO-COW : Protocol Compliance on the Web*. AT&T Labs Technical Report 990803-05-TM, 1999.
http://www.ietf.org/proceedings/99nov/slides/plenary-pro-cow-99nov.pdf

[131] Krishnamurthy, B., Mogul, J. C. and Kristol, D. M. *Key Differences Between HTTP/1.0 and HTTP/1.1*, in *Proceeding of the Eighth International Conference on the World Wide Web*, (Toronto, Canada, 1999), Computer Networks (Elsevier North-Holland, Inc.), pp. 1737-1751.
http://www.research.att.com/~bala/papers/h0vh1.html

[132] Krishnamurthy, B. and Rexford, J. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, 2001. 672pp.

[133] Kubiatowicz, J. *Extracting Guarantees from Chaos* in *Communications ACM*, 2003, *46* (2). pp. 33-38.

[134] Lamport, L. *Time, Clocks and the Ordering of Events in a Distributed System* in *Communications of the ACM*, 1978, *21* (7). pp. 558-565.

[135] Lamport, L. *The Mutual Exclusion Problem: Part I-A Theory of Interprocess Communication* in *Journal of the ACM*, 1986, *33* (2). pp. 313-326.

[136] Lamport, L. *The Mutual Exclusion Problem: Part II-Statement and Solutions* in *Journal of the ACM*, 1986, *33* (2). pp. 327-348.

[137] Lamport, L. *Distribution.* Digital Equipment Corp. Systems Research Center (SRC), 28 May 1987.
http://research.microsoft.com/users/lamport/pubs/distributed-system.txt

[138] Lamport, L. *The Mutual Exclusion Problem Has Been Solved* in *Communications of the ACM*, 1991, *34* (1). p. 110.
http://research.microsoft.com/users/lamport/pubs/lamport-mutual-solved.pdf

[139] Lamport, L., Shostak, R. and Pease, M. *The Byzantine Generals Problem* in *ACM Transactions on Programming Languages and Systems*, 1982, *4* (3). pp. 382-401.

[140] Lawrence, S. and Khare, R. *RFC 2817: Upgrading to TLS Within HTTP/1.1.* IETF, May 2000.

[141] Leivent, J. I. and Watro, R. J. *Mathematical Foundations for Time Warp Systems* in *ACM Transactions on Programming Languages and Systems*, 1993, *15* (5). pp. 771-794.

[142] Luckham, D. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002. 400pp.

[143] Luckham, D. C. *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events*, in *DIMACS Partial Order Methods Workshop IV*, (Princeton University, July 1996).

[144] Lynch, N. A. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1996. 904pp.

[145] Lyons, R. K. *Foreign exchange volume: Sound and fury signifying nothing?* in Frankel, J. A., Galli, G. and Giovanni, A. eds. *The microstructure of foreign exchange markets*. University of Chicago Press, Chicago, 1996, pp. 183-205.

[146] Martin, A. *Wires, Forks, and Multiple-Output Gates* in Gries, D. ed. *Beauty is Our Business*, 1990, p. 304.

[147] Matena, V. and Hapner, M. *Enterprise Java Beans Specification, v1.1*. Sun Microsystems, Inc., Palo Alto, 1999. `ftp://ftp.java.sun.com/pub/ejb/11final-129822/ejb1_1-spec.pdf`

[148] Maurer, H. *HyperWave: The Next-Generation Web Solution*. Addison-Wesley, Harlow, England, 1996.

[149] Medvidovic, N. and Taylor, R. N. *Exploiting Architectural Style to Develop a Family of Applications* in *IEE Proceedings - Software Engineering*, 1997, *144* (5-6 (October/December)). pp. 237-248.

[150] Millard, P. *JEP-0060: Publish-Subscribe*. Jabber Software Foundation, 2003. `http://www.jabber.org/jeps/jep-0060.html`

[151] Miller, G. A. *WordNet: A Lexical Database for English* in *Communications of the ACM*, 1995, *38* (11). pp. 39-41.

[152] Miller, J. *Jabber Instant Messaging Server*, Jabber Software Foundation, 1999. `http://www.jabber.org/`

[153] Miller, J., Saint-Andre, P. and Bamonti, T. *XMPP [eXtensible Messaging and Presence Protocol] CPIM [Common Profile for Instant Messaging] Mapping (experimental)*. IETF, 2002. `http://www.ietf.org/internet-drafts/draft-miller-xmpp-cpim-00.txt`

[154] Mills, D. *RFC 2030: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. University of Delaware, October 1996. `http://www.ntp.org`

[155] Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S. and Xu, Z. *Peer-to-Peer Computing*. HP Labs, Palo Alto, CA, 2002. 51pp.

[156] Mockapetris, P. *RFC 1035: Domain Names - Implementation and Specification*. IETF, 1987. `ftp://ftp.isi.edu/in-notes/rfc1035.txt`

[157] Mogul, J. C. *The Case for Persistent-Connection HTTP*, in *SIGCOMM'95*, (Cambridge, MA, August 1995), ACM, pp. 299-313.

[158] Mogul, J. C. *Errors in Timestamp-based HTTP Header Values.* [Compaq] Western Research Laboratory Research Report 99/3, December 1999.

[159] Mogul, J. C., Douglis, F., Feldmann, A. and Krishnamurthy, B. *Potential Benefits of Delta Encoding and Data Compression for HTTP*, in *SIGCOMM'97*, (Cannes, France, 1997), ACM, pp. 181-194.

[160] Monroe, R. T., Kompanek, A., Melton, R. and Garlan, D. *Architectural Styles, Design Patterns, and Objects* in *IEEE Software*, 1997, *14* (1). pp. 43-52.

[161] Monson-Haefel, R. and Chappell, D. *Java Message Service*. O'Reilly & Associates, 2000. 238pp.

[162] Moore, K. *RFC 3205: BCP 56: On the use of HTTP as a Substrate.* Internet Engineering Task Force, February 2002.

[163] Myers, J. *RFC 2086: IMAP4 ACL Extension.* Internet Engineering Task Force, January 1997.

[164] Nagle, J. *RFC 896: Congestion Control in IP/TCP Internetworks.* IETF, 6 January 1984. `ftp://ftp.isi.edu/in-notes/rfc896.txt`

[165] Nardi, B., Whittaker, S. and Bradner, E. *Interaction and Outeraction: Instant Messaging in Action*, in *ACM Conference on Computer Supported Cooperative Work (CSCW 2000)*, (Philadelphia, PA, December 2000), ACM Press, pp. 79-88. `http://www.acm.org/pubs/articles/proceedings/cscw/358916/p79-nardi/p79-nardi.pdf`

[166] Neumann, A. L. *The Great Firewall.* Committee to Protect Journalists, New York, NY, January 2001. 8pp. `http://www.cpj.org/Briefings/2001/China_jan01/Great_Firewall.pdf`

[167] Neumann, P. G. *Computer-Related Risks*. ACM Press/Addison-Wesley, 1995. 384pp.

[168] Norris, R. *JEP-0021: Jabber Event Notification Service*. Jabber Software Foundation, 2002. `http://www.jabber.org/jeps/jep-0021.html`

[169] O'Neil, P. *The Escrow Transactional Method* in *ACM Trans. on Database Systems (TODS)*, 1986, *11* (4). pp. 405-430.

[170] O'Ryan, C., Schmidt, D. C. and Noseworthy, J. R. *Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations* in *International Journal of Computer Systems Science and Engineering*, 2002, *17*.

[171] Object Management Group. *CORBAservices: Common Object Services Specification.* July 1997.

[172] Object Management Group (ed.), *The Common Object Request Broker: Architecture and Specification* Object Management Group, 2001. 946pp.
http://www.omg.org/cgi-bin/doc?formal/01-02-33

[173] Office of Energy Efficiency and Renewable Energy. *Strategic Plan for Distributed Energy Resources.* U.S. Department of Energy, September 2000. 34pp.
http://www.eere.energy.gov/der/pdfs/derplanfinal.pdf

[174] Oki, B. M., Pflugl, M., Siegel, A. and Skeen, D. *The Information Bus: An Architecture for Extensible Distributed Systems*, in *Sixteenth Symposium on Operating Systems Principles*, (1993), pp. 58-68.

[175] Olson, M. A., Bostic, K. and Seltzer, M. I. *Berkeley DB*, in *USENIX Annual Technical Conference (FREENIX track)*, (Monterey, California, 6-11 June 1999), pp. 183-191. http://www.sleepycat.com/

[176] Oram, A., Minar, N. and Shirky, C. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies* 1st ed. O'Reilly & Associates, 2001. 432pp.

[177] Oreizy, P. *Decentralized Software Evolution*, in *International Conference on the Principles of Software Evolution (IWPSE 1)*, (Kyoto, Japan, April 20-21 1998).

[178] Patterson, D. A., Gibson, G. and Katz, R. H. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, in *International Conference on Management of Data*, (Chicago, IL, 1-3 June 1988), ACM SIGMOD, pp. 109-116.

[179] Perry, D. E. and Wolf, A. L. *Foundations for the Study of Software Architecture* in *ACM SIGSOFT Software Engineering Notes*, 1992, *17* (4). pp. 40-52.

[180] Petersen, G. L. *Myths about the mutual exclusion problem* in *Information Processing Letters*, 1981, *12* (3). pp. 115-116.

[181] Postel, J. B. *RFC793: Transmission Control Protocol.* IETF, September 1981. 85pp.

[182] Postel, J. B. *RFC 821: Simple Mail Transfer Protocol.* IETF, August 1982.

[183] Ramduny, D., Dix, A. and Rodden, T. *Getting to Know: The Design Space for Notification Servers*, in *Computer Supported Cooperative Work (CSCW'98)*, (Seattle, Washington, November 1998 1998).
http://www.comp.lancs.ac.uk/computing/users/dixa/papers/GtK98/GtK98-full.html

[184] Ranadive, V. *The Power of Now: How Winning Companies Sense and Respond to Change Using Real-Time Technology* Hardcover ed. McGraw-Hill Osborne Media, 1999. 214pp.

[185] Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Shenker, S. *A Scalable Content Addressable Network*, in *Proceedings of ACM SIGCOMM*, (San Diego, CA, 27-31 August 2001), pp. 161-172.

[186] Reed, D. P. *The End of the End-to-End Argument*. 2000.
http://www.reed.com/Papers/endofendtoend.html

[187] Renesse, R. v., Birman, K. P. and Maffeis, S. *HORUS: A Flexible Group Communication System* in *Communications of the ACM*, 1996, *39* (4). pp. 76-83.

[188] Rescorla, E. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000. 499pp.

[189] Rifkin, A. *A Survey of Event Systems.* July 1998.
http://www.ifindkarma.com/attic/isen/event-systems.html

[190] Rifkin, A. and Khare, R. *A Bibliography of Event Papers.* July 1998.
http://www.ifindkarma.com/attic/isen/event-papers.html

[191] Rifkin, A. and Khare, R. *The Evolution of Internet-Scale Event Notification Services: Past, Present, and Future.* (Manuscript), 10 August 1998.
http://www.ifindkarma.com/attic/isen/wacc/

[192] Rivest, R. and Lampson, B. *SDSI - A Simple Distributed Security Infrastructure*, in *CRYPTO'96*, (Santa Barbara, CA, 1996).
http://theory.lcs.mit.edu/~rivest/sdsi11.html

[193] Rodrigues, R., Castro, M. and Liskov, B. *BASE: Using Abstraction to Improve Fault Tolerance*, in *Proceedings of the 18th ACM Symposium on Operating System Principles*, (Banff, Canada, October 2001), pp. 15-28.
http://www.pmg.lcs.mit.edu/~rodrigo/base.pdf

[194] Rose, M. T. *BEEP: The Definitive Guide*. O'Reilly, Sebastopol, CA, 2002. 240pp.

[195] Rosenblum, D. S., Redmiles, D. F. and Robbins, J. E. *Modeling Software Architectures in the Unified Modeling Language* in *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2002, *11* (1).

[196] Rosenblum, D. S. and Wolf, A. L. *A Design Framework for Internet-Scale Event Observation and Notification*, in *6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Zurich, Switzerland, September 1997), Springer-Verlag, pp. 344-360.

[197] Rosenblum, D. S. and Wolf, A. L. *Internet Scale Event Notification*, in *Workshop on Internet Scale Event Notification (WISEN'98)*, (Irvine, CA, July 13-14 1998).
http://www.ics.uci.edu/~irus/wisen/wisen98/abstracts/abs_rosenblum.html

[198] Saltzer, J. H., Reed, D. P. and Clark, D. D. *End-to-End Arguments in System Design.* 2000. http://www.reed.com/Papers/endtoend.html

[199] Samuelson, P. A. and Nordhaus, W. D. *Economics* 17th ed. McGraw-Hill, 2001.

[200]Schneier, B. *Secrets and Lies*. John Wiley & Sons, Inc., 2000. 432pp.

[201] Shannon, C. E. *A Mathematical Theory of Communication* in *Bell System Technical Journal*, 1948, *27* (3 & 4). pp. 379-423 & 623-656.

[202] Shaw, M. *The Coming-of-Age of Software Architecture Research*, in *the 23rd International Conference on Software Engineering*, (Toronto, Canada, 2001), IEEE Computer Society, pp. 656-664.

[203] Shaw, M. and Clements, P. *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*, in *Computer Software and Applications Conference*, (August 1997), pp. 6-13.

[204] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M. and Zelesnik, G. *Abstractions for Software Architecture and Tools to Support Them* in *IEEE Transactions on Software Engineering*, 1995, *21* (4). pp. 314-335.
`http://citeseer.nj.nec.com/shaw95abstractions.html`

[205] Siegel, J. *CORBA Fundamentals and Programming*. Wiley, New York, NY, 1996.

[206] Singhal, S. and Cheriton, D. *Using projection aggregations to support scalability in distributed simulation*, in *International Conference on Distributed Computing (ICDCS'96)*, (1996).

[207] Sontag, E. D. *Mathematical Control Theory: Deterministic Finite Dimensional Systems* 2nd ed. Springer Verlag, 1998. 531pp.
`http://www.math.rutgers.edu/~sontag/FTP_DIR/mct-intro.pdf`

[208] Spira, J. B. *20 Years—One Standard: The Story of TCP/IP* in *Iterations: An Interdisciplinary Journal of Software History*, 2003 (2). pp. 1-3.
`http://www.cbi.umn.edu/iterations/spira.html`

[209] Srisuresh, P. and Egevang, K. *RFC 3022: Traditional IP Network Address Translator (Traditional NAT)*. Internet Engineering Task Force, January 2001.

[210] Stein, L. D. *Web Security: A Step-By-Step Reference Guide*. Addison-Wesley, 1997. 448pp.

[211] Stuurman, S. and van Katwijk, J. *Evaluation of Software Architecture for a Control System: A Case Study*, in *Proceedings of the Second International Conference on Coordination Models and Languages*, (Berlin, Germany, September 1997), pp. 157-171.
`http://www.ou.nl/open/stm/publikaties/coordination97.pdf`

[212] Sutherland, I. E. and Ebergen, J. *Computers Without Clocks* in *Scientific American*, August, 2002. vol. *287* (2).

[213] Taylor, R. N., Medvidovic, N., Anderson, K. M., E. James Whitehead, J., Robbins, J. E., Nies, K. A., Oreizy, P. and Dubrow, D. L. *A Component- and Message-Based Architectural Style for GUI Software* in *IEEE Transactions on Software Engineering*, 1996, *22* (6). pp. 390-406.

[214] Tennenhouse, D. *Active Networks*, in *Second Symposium on Operating Systems Design and Implementation*, (Seattle, WA, October 1996), USENIX Association, p. 89.

[215] Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J. and Minden, G. J. *A Survey of Active Network Research* in *IEEE Communications Magazine*, 1997, *35* (1). pp. 80-86.

[216] Thomas, S. *Summerlin v. Stewart (Case #98-99002).* United States Court of Appeals for the Ninth Circuit, September 3 2003.
http://www.ca9.uscourts.gov/ca9/newopinions.nsf/C9D33A3EF8BC665C88256D95005BFD72/$file/9899002.pdf

[217] Thornley, J. *A Parallel Programming Model with Sequential Semantics* (PhD Thesis), California Institute of Technology, Computer Science Department, 1996.

[218] Touch, J. D. *Mirage: A Model for Latency in Communication* (PhD Thesis), University of Pennsylvania, Computer and Information Science, 1992.

[219] Touch, J. D. *RFC 2140: TCP Control Block Interdependence.* Internet Engineering Task Force, April 1997.

[220] Udell, J., Gillmor, S. and Eds. *2002 Technology of the Year Award: Publish/Subscribe Technology: KnowNow 1.5* in *Infoworld*, January 24, 2003.
http://www.infoworld.com/article/03/01/24/2002TOY-sb_1.html

[221] University of California, B. *UNIX Programmer's Manual, 4.3 Berkeley Software Distribution*. Computer Science Division, University of California, Berkeley, 1986.

[222] US Department of Justice. *Emulex Hoaxer Indicted for Using Bogus Press Release and Internet Service to Drive Down Price of Stock* in *Press Release*, September 28, 2000. http://www.cybercrime.gov/emulex.htm

[223] van de Snepscheut, J. L. A. *What Computing Is All About*. Springer-Verlag, 1993.

[224] van de Snepscheut, J. L. A. *The sliding-window protocol revisited* in *Formal Aspects of Computing*, 1995, *7* (1). pp. 3-17.
http://resolver.caltech.edu/CaltechCSTR:1991.cs-tr-91-06

[225] van Rossum, G. and Drake, F. L. *An Introduction to Python*. Network Theory, 2003. 120pp.

[226] von Neumann, J. and Morgenstern, O. *Theory of Games and Economic Behavior*. Princeton University Press, 1944. 625pp.

[227] Waldo, J. *A Minimalist Approach to Distributed Event Notifications*, in *Workshop on Internet Scale Event Notification (WISEN'98)*, (Irvine, CA, 13-14 July 1998).
http://www.ics.uci.edu/~irus/wisen/wisen98/abstracts/abs_waldo.html

[228] Wall, L., Christiansen, T. and Schwartz, R. L. *Programming Perl* 2nd ed. O'Reilly & Associates, 1996.

[229] Whitehead, E. J., Robbins, J. E., Medvidovic, N. and Taylor, R. N. *Software Architecture: Foundation of a Software Component Marketplace*, in *Proc. First International Workshop on Architectures for Software Systems*, (New York, 1995), ACM, pp. 276-282.

[230] Whitehead, E. J. and Wiggins, M. *WEBDAV: IETF Standard for Collaborative Authoring on the Web* in *IEEE Internet Computing*, September/October, 1998. pp. 34-40.

[231] Williams, S. and Kindel, C. *The Component Object Model: A Technical Overview.* Microsoft Corporation, 1994.
http://msdn.microsoft.com/library/techart/msdn_comppr.htm

[232] Wirth, T. E., Gray, C. B. and Podesta, J. D. *The Future of Energy Policy* in *Foreign Affairs*, 2003, *82* (4). pp. 132-155.
http://www.foreignaffairs.org/20030701faessay15410/timothy-e-wirth-c-boyden-gray-john-d-podesta/the-future-of-energy-policy.html

[233] Wood, L. and et al. *REC-DOM-Level-1: Document Object Model (DOM) Level 1 Specification.* World Wide Web Consortium, Cambridge, MA, October 1998.
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001

[234] Working Group C3 - Electric Network Control Systems Standards. *C37.1: Standard Definition, Specification, and Analysis of Systems Used for Supervisory Control, Data Acquisition, and Automatic Control.* IEEE/ANSI, 1994.
http://grouper.ieee.org/groups/sub/wgc3/c371_mn.htm

[235] Zimmerman, H. *OSI Reference Model: The ISO Model of Architecture for Open Systems Interconnection* in *IEEE Transactions on Communications*, 1980, *COM-28* (4). pp. 425-432.