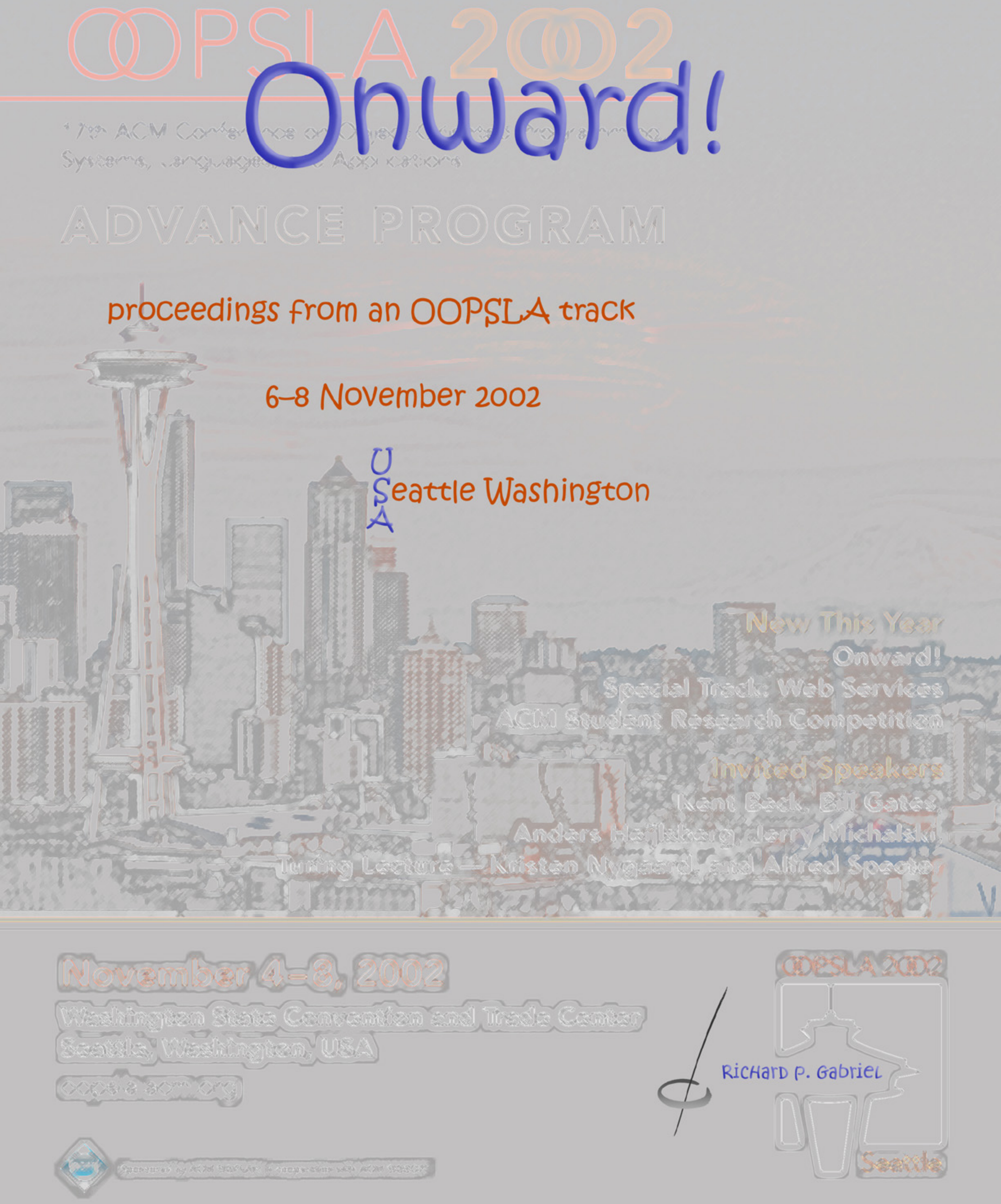
Onward!

proceedings from an OOPSLA track

6–8 November 2002

USA Seattle Washington

Richard P. Gabriel


OOPSLA 2002
17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications
ADVANCE PROGRAM
New This Year
Onward!
Special Track: Web Services
ACM Student Research Competition
Invited Speakers
Kent Beck, Bill Gates,
Anders Hejlsberg, Jerry Michalski,
Turing Lecture — Kristen Nygaard, and Alfred Spector
November 4–8, 2002
Washington State Convention and Trade Center
Seattle, Washington, USA
oopsla.acm.org
Sponsored by ACM SIGPLAN in cooperation with ACM SIGSOFT
OOPSLA 2002
Seattle

Onward!

A Track at OOPSLA 2002

Seattle, Washington

November 6–8, 2002

The Onward! Track contains technical and philosophical papers describing new paradigms or metaphors in computing, new thinking about objects, new framings of computational problems or systems, and new technologies. Papers in the Onward! Track aren't aimed at advancing the state of the art—they're aimed, instead, at altering or redefining the art by proposing a leap forward—or sideways—for computing.

Chair

Richard P. Gabriel
Sun Microsystems, Inc.

Program Committee

Geoff Cohen
Cap Gemini Ernst & Young

Pierre Cointe
L'Ecole des Mines de Nantes

Dave Thomas
Bedarra Corporation and Carleton University

Dream Songs Press 2002

The Onward! Program

Wednesday, 6 November 2002 13:30–15:00

Convention Center—Ballroom 6D

New Models for Software I

This session sets the stage for the Onward! track by presenting two extreme points in the discussion on how to move forward with software: Autonomic computing, at one end, is being both researched and designed into existing systems today, while the metaphor of magic is intended to stimulate people to think far beyond the walls of the box and the academy.

A Vision of Autonomic Computing

Jeffrey O. Kephart
IBM
kephart@us.ibm.com

Autonomic Computing is a new approach to coping with the rapidly growing complexity of operating and integrating computing systems. This paper elaborates on the vision of autonomic, or self-managing computing. The goal is not to present a well-articulated architecture. Instead, the author speculates about general architectural features that might typify autonomic computing systems of the future, and uses this as a basis for discussing several challenging scientific and engineering issues that need to be addressed in order to realize the vision of autonomic, or self-managing computing.

Magic

Dave West
New Mexico Highlands University
dwest@cs.nmhu.edu

Assume an obverse version of Clarke's insight: "If a technology is not magical, it is insufficiently advanced." Computing and software development are clearly not magical even though some applications, especially in cinema special effects, certainly convey magical impressions. The central question of this essay—can we use magic as a metaphor to re-evaluate and redefine the theory and practice of computing? Or, stated slightly differently, can magic provide a metaphor for opening a new frontier in the investigation and solution of the core problems confronted by software developers and computing professionals in today's world?

Wednesday, 6 November 2002 15:30–17:00
Convention Center—Ballroom 6D

Panel: Biologically Inspired Software

Steven Hofmeyr
Company 51
steve@company51.com

Jeffrey O. Kephart
IBM
kephart@us.ibm.com

Dave West
New Mexico Highlands University
dwest@cs.nmhu.edu

Both biologists and computing researchers need metaphors and concepts to understand how complex, large systems work, even if that understanding is merely statistical or otherwise non-predictive. This panel explores the question of furthering computing and organizing computations by looking at the concepts found in biology, used by biologists, and required by biologists.

Wednesday, 6 November 2002 19:30–21:00

Convention Center—Exhibit Hall 4B

Keynote: What's Next?

Jerry Michalski
Sociate

Technology is not neutral. It reflects the objectives and mental models of those who design it, the business imperatives of the times and the interactions of those who use it. By tracing the history behind some of today's critical technologies, then describing the dynamics between the major forces in the business market and the market of ideas, Jerry will tackle questions such as:

- Why does it seem that innovation is at a standstill, despite much emphasis on corporate innovation?
- What role do our assumptions about capitalism, intellectual property, assets and scarcity have in our continuing evolution?
- How will programming quickly/slowly, in the large/in the small, with closed/open models and with highly structured/unstructured, organic approaches play out?
- Where should technology developers place their energies?

Thursday, 7 November 2002 10:30–12:00
Convention Center—Ballroom 6D

New Models for Software II

This session looks at new ways of thinking about software and how to build it, through the lenses of post-modernism and just-in-time manufacturing processes. Both these papers are concerned with how people deal with software, not simply with its technologies.

Notes on Postmodern Programming

James Noble
Victoria University of Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Robert Biddle
Victoria University of Wellington, New Zealand
robert@mcs.vuw.ac.nz

What is postmodern computer science? How is programming related to computer science? The authors have written, "Let us create a new guild of programmers without the class-distinctions that raise an arrogant barrier between programmers and computer scientists! Let us desire, conceive, and create the new program of the future together. It will combine design, user-interfaces, and programming in a single form, and will one day rise towards the heavens from the hands of a million workers as the crystalline symbol of a new and coming faith."

Principles of Lean Thinking

Mary Poppendieck
Poppendieck.LLC
mary@poppendieck.com

In the 1980's, a massive paradigm shift hit the factories throughout the US and Europe. Mass production and scientific management techniques from the early 1900's were questioned as Japanese manufacturing companies demonstrated that "Just-in-Time" was a better paradigm. The widely adopted manufacturing concepts came to be known as "Lean Production". When appropriately applied, Lean Thinking is a well-understood and well-tested platform upon which to build agile software development practices.

Thursday, 7 November 2002 13:30–15:00
Convention Center—Ballroom 6D

New Programming Constructs

This session presents two practical ideas derived from challenging common assumptions. The first is the result of questioning the assumption that a server is necessary to coordinate cooperating networks, and the second questions a commonsense design rule.

Many-to-Many Invocation

Alan Kaminsky
Rochester Institute of Technology
ark@cs.rit.edu

Hans-Peter Bischof
Rochester Institute of Technology
hpb@cs.rit.edu

Many-to-Many Invocation (M2MI) is a new paradigm for building collaborative systems that run in wireless proximal ad hoc networks of fixed and mobile computing devices. M2MI is useful for building a broad range of systems, including multiuser applications (conversations, groupware, multiplayer games); systems involving networked devices (printers, cameras, sensors); and collaborative middleware systems.

Problematic Encapsulation in High-Risk Systems

Daniel Dvorak
Jet Propulsion Laboratory, California Institute of Technology
daniel.dvorak@jpl.nasa.gov

One of the most common metaphors in OOAD clashes with the physics of the real world. Moreover, this clash isn't obvious in everyday systems—it only becomes obvious in a category of systems called "high risk systems." The metaphor is that of designing an object model that is isomorphic to the hardware aggregation hierarchy, i.e., decomposition by subsystem and device, with encapsulated state. Hardware units seem like obvious candidates for objects; the paper shows how this 'obvious' metaphor breaks down and can lead to a messy design. The paper uses examples from NASA space missions involving control of spacecraft and Mars rovers as examples of high-risk systems.

Thursday, 8 November 2002 12:00–13:00
Convention Center—Rooms 602–604

Panel: New Programming Constructs Beyond Inheritance, Patterns, and Notation: What's left?

Geoff Cohen
Cap Gemini Ernst & Young Center for Business Innovation
geoff.cohen@cgey.com

William R. Cook
Allegis Corporation
william@allegis.com

Robert Filman
Ames Research Center, NASA
rfilman@mail.arc.nasa.gov

There hasn't been much beyond incremental improvements to programming languages for about the last 10 years. We see implementation advances, variants on encapsulation and composition, notation, process, patterns, and a steady diet of types, types, and more types. This panel tries to uncover what we haven't thought of yet.

The Onward! Papers

A Vision of Autonomic Computing

Jeffrey O. Kephart
IBM Research
Yorktown Heights, NY 10598
kephart@us.ibm.com

October 29, 2002

Abstract

In October, 2001, IBM released a manifesto [1] that defined *autonomic computing*, a new approach to coping with the rapidly growing complexity of integrating, configuring and operating computing systems. This article elaborates further on IBM's vision of autonomic computing. We speculate about general architectural features that might typify autonomic, or self-managing, systems of the future, and use this as a basis for discussing several challenging scientific and engineering issues that need to be addressed in order to realize this vision. Our goal is to motivate the academic and industrial research communities to address these fundamental problems.

1 Introduction

On March 8, 2001, Paul Horn, Senior Vice President of IBM Research, announced IBM's commitment to *autonomic computing*—a fundamentally new approach to coping with the rapidly growing complexity of operating and integrating computing systems [2]. This was soon followed in April of that year by the announcement of Project eLiza, an initiative to build self-managing capabilities into IBM supercomputers and mainframes. Then, on October 15, 2001, IBM released a manifesto that justified the need for autonomic computing, and described in broad terms several characteristics that autonomic computing systems of the future will possess [1]. The manifesto observed that the main obstacle to further progress in the information technology industry is not a slowdown in Moore's law. Rather, it is the inexorability of Moore's law that has led us to the verge of a software complexity crisis. Over the last few decades, programmers have fully exploited a four- to six-orders-of-magnitude increase in computational power, producing ever more sophisticated and functional software applications and environments—some weighing in at tens of millions of lines of code, and requiring skilled I/T professionals to install, configure, tune, and maintain them.

But the difficulty of managing today's computing systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into corporate-wide computing systems introduces a whole new level of complexity. With the rapid growth of the Internet and electronic commerce, interconnectedness is now being extended beyond company boundaries by businesses seeking to integrate their business processes with those of their trading partners, compounding the problems of system management and integration to the point where it can take several person-years of effort. We appear to be approaching the limits of human capability, yet the march towards increased interconnectivity and integration appears to be rushing ahead unabated. Complexity could turn the dream of pervasive computing, with its vision of trillions of computing devices connected to the Internet [3], into a nightmare.

Past crises in software complexity have often been overcome by programming language innovations, such as high-level languages, structured programming, and object-oriented programming, that have extended the size and complexity of systems that architects can design. But sole reliance on further innovations in programming methods will not get us though the present crisis. As systems are becoming ever more interconnected with an increasingly diverse set of other systems and environments, architects are less and less able to pre-plan intricate interactions among system components, leaving such issues to be resolved at run time. This places on system administrators and system integrators a burden that neither are capable of assuming, due to a growing labor shortage[1] and the growing gap between the inherent difficulty of system management and the fixed limits of human capability. If we continue on our present course, installing, configuring, optimizing, maintaining, and merging massive, complex, and heterogeneous computing systems will become too difficult for even the most skilled I/T workers, as will the task of making sufficiently quick, decisive reponses to rapid streams of changing and conflicting demands.

We are left with only one option: to create computing systems that manage themselves. More specifically, we must create computing systems that configure, heal, optimize and protect themselves in accordance with high-level behavioral specifications from human administrators.

IBM has coined the phrase *autonomic computing* to represent its vision of how the world will rise to this new grand challenge. According to the Oxford English Dictionary, there is no essential difference between the primary definitions of "autonomic" and "autonomous." Both words mean "self-governing." But by choosing *autonomic*, we are deliberately playing on the biological connotation. The *autonomic nervous system* governs many of our involuntary functions, including our heart rate, our respiratory rate, our blood's sugar and oxygen levels, our body temperature, our digestion, and the dilation of our pupils. It frees our conscious brain from the burden of having to deal with these vital but lower-level functions.

The autonomic nervous system is emblematic of many complex natural systems that are self-governing. In essence, life on Earth is a concentric series of levels of aggregation, each level comprising many interacting, autonomous, self-governing components, each of which may be composed of large numbers of interacting, autonomous, self-governing components at the next level down. Consider the enormous range in

[1]The I/T labor shortage in the United States alone is expected to grow from roughly 450,000 in 2001 to 600,000 in 2002, despite the weak economy [4].

scale that starts with molecular machines within cells, then up to individual cells, increasingly complex multi-cellular organisms, social groups such as hives and herds, and beyond to the entire world ecosystem. In the case of humans, parallel branches of this hierarchy extend beyond individuals to tribes, societies or markets, and beyond these to the entire world socio-economy. It is entirely within the purview of autonomic computing to seek inspiration in social and economic systems as well.

Thus, in our use of the term *autonomic computing*, we are really expressing both a desire to find new paradigms for achieving self-governance of massive and complex computing systems *and* a belief that inspiration for those new paradigms can be found in a variety of natural systems, of which the autonomic nervous system is representative. We are encouraged by prior successes in applying biological and economic principles to problems in computer science, including anti-virus [5] and intrusion detection systems [6] patterned after the immune system, adaptive decentralized network routing algorithms inspired by the social intelligence of ants [7], and market-based distributed computation and problem solving approaches [8].

In its original manifesto, IBM set forth the grand challenge of creating self-managing computing systems, and suggested that nature may hold the key to their design. But noone believes that this challenge can be met by any one organization. It will take a concerted, long-term, worldwide effort by researchers in diverse fields to meet this challenge. The purpose of this article is to paint a somewhat more detailed picture of what autonomic computing systems might look like, discuss in general terms how they might function, and to use this as a basis for outlining some of the major challenges that we face in designing them and understanding their behavior. Specifically, in section 2 we elaborate on the notion of self-management, and present some scenarios that illustrate the behavioral characteristics that will typify autonomic computing systems. Then, in section 3, we discuss a set of architectural considerations that serve as the basis for discussions in sections 4 and 5 of some fundamental engineering and scientific issues that need to be tackled if we are to realize our vision of autonomic computing.

2 Perspectives on Self-Management

The essence of autonomic computing systems is self-management. This section presents a few different perspectives on the nature of self-management, beginning with present-day views, then speculating about how the notion of self-management may evolve over the coming years, and finally culminating in a series of vignettes that depicts how people may experience autonomic systems several years hence.

Ever since the launch of its autonomic computing initiative, IBM has been citing four aspects of self-management: *self-configuration*, *self-healing*, *self-optimization*, and *self-protection*. This view of self-managment naturally reflects a concern with specific present-day problems that make system administration particularly difficult and burdensome. Let us expand a bit upon each of these facets of self-management:

- **Self-configuration.** Today, installing, configuring and integrating large complex systems can be quite challenging, time-consuming, and error-prone—even for experts. Large web sites or corporate data centers are typically a haphazard accretion of servers, routers, databases, and other technologies on several different platforms from several different vendors, the full configuration and functionality of which cannot be grasped by any single human mind. It can take months of effort by teams of expert programmers to make significant changes to such systems (such as installing a major e-commerce application such as SAP, or merging two such systems into one).

Autonomic systems will be able to configure themselves automatically in accordance with high-level policies (representing business-level objectives, for example) that specify what is desired, not how it is to be accomplished. Adding a new component will be like adding a new cell to the body, or a new individual to a large population. When it is introduced to the system, it will incorporate itself seamlessly (even if it is of a new type), and the rest of the system will adapt to its presence automatically, to the mutual benefit of the system and the new component.

- **Self-optimization.** Today, large complex middleware (e.g. WebSphere) or database (e.g. Oracle or DB2) systems have dozens or even hundreds of tunable parameters. The performance of the system is strongly and nonlinearly dependent on the settings of these parameters, and is also strongly dependent on the usage conditions, so that an untuned system with parameters set to their default values can perform very poorly. Retuning to keep up with changes in usage patterns is very daunting, and therefore undertaken much less frequently than would be optimal. The complexity of database tuning, for example, is indicated by the number of books written on the topic—a quick survey of online bookstores turns up 24 on Oracle database tuning alone [9, 10]. New books have to be written continually, as the performance metrics and the tuning knobs change with each release level of the same product. One author [11] has commented that "If you write a technical book about Oracle, it will be out of date by the time you've finished writing it, and within a year of publication it will be 20% misleading, inappropriate, or just plain wrong." The same holds for any system of comparable complexity and functionality. As if this were not bad enough, consider that such systems are often integrated with one another, and therefore performance tuning of one large subsystem can have unanticipated effects on the system as a whole.

  Autonomic systems will continually seek ways to improve their operation, identifying and seizing opportunities to make themselves more efficient in terms of performance or cost—just as muscles become stronger through exercise, and the brain modifies its circuitry during the course of learning. They will achieve this by monitoring, experimenting with, and tuning their own parameters, and by making appropriate choices about insourcing or outsourcing functions.

- **Self-healing and self-protection.** Today, problem determination is a huge, important, and difficult enterprise. IBM and other I/T vendors have large departments devoted to identifying, tracing, and determining the root cause of errors and failures in large, complex computing systems. The most serious customer problems can sometimes take teams of programmers several weeks to diagnose and fix, and sometimes the problem disappears mysteriously after a sufficient amount of trial and error, without any satisfactory diagnosis ever being made.

Autonomic computing systems will be self-healing—capable of detecting, diagnosing, and repairing localized problems arising from bugs or failures in software or hardware. They will also be self-protecting in at least two different senses. First, they will defend the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures. Second, they will anticipate potential problems (perhaps based on early reports from sensors or components) and take steps to avoid them, or at least to mitigate their effect.[2]

In early versions of autonomic systems, these four aspects of self-management may be treated as distinct from one another, with different product teams creating individual solutions that address each issue separately. Ultimately, however, we believe that autonomic systems will be built in such a way that these aspects of self-management will be emergent properties of a general autonomic architecture. As a consequence, the present-day distinctions among these properties will begin to blur. For example, self-protection, self-healing, and some aspects of self-optimization may well merge into more encompassing notions of *self-maintenance* and *robustness*:

- **Self-maintenance and robustness.** In a manner analogous to that of their biological namesakes, autonomic systems will maintain and adjust their operation in the face of changing workloads, demands and external conditions, and in the face of hardware or software failures of innocent or malicious origin. Thus self-healing and self-protection may simply be more extreme manifestations of the system's continual efforts to accommodate and adapt to change. Autonomic systems will display another aspect of self-maintenance—proactively seeking to upgrade their function by finding, verifying and applying the latest software updates.

Another trend that we foresee is the gradual adoption of automation in autonomic systems over time. The usual historical pattern of automation will be followed. At first, automated functions will merely collect information, and help aggregate it in ways that support decisions by human administrators. Later, they will serve in an advisory capacity, suggesting possible courses of action, and offering to execute these suggested actions at the press of a button. As the automation technologies improve, and as humans' faith in those technologies grows in tandem (perhaps lagging a year or two behind), humans will entrust autonomic systems with making (and acting

[2]Somayaji and Forrest[12] have developed an interesting example of automated protection called *process homeostasis*, in which abnormal Linux processes are delayed in proportion to the degree of their abnormality. This technique slows down processes that are potentially damaging to the system, helping it to cope with malicious attacks and propagating failures.

upon) more and more of the lower-level decisions. In effect, the roles will be reversed: humans will now serve as advisors to the autonomic system, rendering relatively less frequent and higher-level decisions that are carried out automatically via more numerous, lower-level decisions and actions taken by the system itself. Over time, human input will be ever more high-level and infrequent.

Thus far, we have focussed mainly on the benefits of autonomic computing to system administrators. In order to portray how end users may experience self-management in relatively sophisticated autonomic systems of the future, we conclude this section with a series of vignettes. The vignettes focus on the higher levels of the autonomic computing hierarchy—from departmental computing systems ranging up to the automated e-business interactions among e-businesses—simply because this is where people will experience autonomic systems most directly.

Now let us listen in on some conversations among co-workers during an ordinary day at MachineCuisine.biz…

9am, by the coffee machine

> **Kyle**: The new mainframe and a couple new servers arrived yesterday afternoon, so I plugged them in.
>
> **Kayla**: Ah, so *that's* why the system seems a little faster this morning!

11am, in Tyler's office

> **Tyler**: I'm worried that I won't be able to finish my Talking Toaster-Roaster design by this afternoon. My Designer app has been *really* slow this morning.
>
> **Taylor**: Hmmm… I heard that Kyle plugged in some new machines last night. Maybe some of the basic system services migrated to the new hardware or upgraded themselves. I bet your app is just a little out of sync right now—it probably needs to be reconfigured or upgraded or something. Did you ask it what's wrong?
>
> **Tyler**: Oh yeah, let's see…right—my app says it wants me to let it upgrade one of its components and re-tune itself. Oh, look—here's a message it sent me last night asking for permission. Arrggh! Why didn't I read my e-mail?
>
> **Taylor**: Geez—you're still using *e-mail* notification? Whatever—just click on *OK*.
>
> **Tyler**: I *know*, Taylor—I've done this *plenty* of times before. Al…right, it says it's done. Let's see. Yup, it's back to normal—actually, it seems a lot faster than before. Cool! This'll put the Talking Toaster-Roaster back on track!
>
> **Taylor**: Great. Umm… why don't you change the software upgrade option in your personal profile from *manual* to *automatic*?

Tyler: I like to wait a couple days to make sure there aren't problems. Wasn't there some kind of bug last year?

Taylor: Oh, you're probably thinking of the one that happened when they put in the new payroll system. That was no big deal. The automatic regression tester discovered the discrepancy a few minutes after the new system was installed, so the old system kicked back in immediately and ran for a few hours until they fixed the bug. I'm surprised you remember...

Lunchtime, in the cafeteria

Britney: Does that burrito really need a whole bottle of tabasco sauce?

Whitney: It only *looks* like a burrito—it sure doesn't *taste* like one. So anyway...did you hear the good news? Our department is probably going to meet its budget after all.

Britney: You're kidding! I thought the earthquake blew our database services costs.

Whitney: It did, but last week another provider opened a new 20-acre database services facility in Canada. A bunch of our databases decided to migrate there. Prices have already dropped below last month's levels, and we're even seeing slightly better response times.

Britney: Terrific!

Whitney: Now if we could only get our chef to migrate to Canada...

3pm, by the coffee machine

Kyle: SAP announced a major upgrade this morning.

Kayla: Yeah, I heard. How's it going?

Kyle: Oh, fine. They've run our acceptance suite and the bleeding edge users have already been switched over. Everyone else will follow their usual switchover profiles.

Kayla: [*Snickering.*] It'll probably take Tyler an hour to press all those *OK* buttons.

Kyle: Maybe just 10 minutes—his fingers have had *lots* of practice!

7pm, at a local restaurant

Tyler: My boss loves the Toaster-Roaster—thanks a lot for helping me this morning.

Taylor: No problem—glad she likes it. Hey—did you hear that some hackers broke into our eFrigerator Service today and deleted all of the FridgeWidgets?

Tyler: Oh no—I bet our customers are pretty mad! And we'll probably have to pay Food Emporium and Safeway some pretty stiff service outage penalties.

Taylor: Not really—they hardly even noticed. Our sentinels immediately noticed that the FridgeWidgets weren't responding, so they told our backup e-utility to handle the load for a few minutes while a fresh set of FridgeWidgets looked around for a new e-utility provider. They found one with a good reputation, and they negotiated a great price and service guarantee.

Tyler: At least for the next couple weeks, until the new provider tries to raise its prices on us!

Taylor: Don't worry—our FridgeWidgets are pretty hard-nosed, and they can always find a cheaper e-utility if they have to.

These vignettes[3] illustrate several aspects of self-management that will be common to autonomic systems and their elements at all levels, from software or hardware components to enterprises to electronic markets: self-upgrading, self-optimizing, and self-healing applications, self-migrating databases, self-installing mainframes and servers, and enterprises that automatically and seamlessly outsource their business processes.

As we have stated, we believe that ultimately these common aspects of self-management will emerge from a common set of architectural principles and concepts that will apply broadly across many different types of applications, and at many different levels, ranging from individual computational elements like disk drives to entire automated businesses that sell information goods and services to one another. The next section provides a broad outline of some basic architectural principles that appear capable of providing a basis for autonomic systems.

3 Architectural Considerations

Thus far, we have painted a picture of how autonomic computing systems might behave, but we have given only vague hints as to how they might be built. In this section, we describe in the most generic terms some architectural thoughts and design patterns that appear promising to us. Our primary purpose is not to define a detailed architecture, but to provide the necessary framework for discussing what we believe to be some of the most important and critical engineering and scientific challenges that need to be addressed by the research community.

We do not believe that, in their ultimate form, autonomic computing systems will be built from separately engineered self-configuration, self-healing, self-optimization,

[3]The vignettes are admittedly unrealistic, in the sense that the true measure of success for autonomic systems will lie in their *unremarkability*: no one will comment upon or even know what they are doing—just as today no one knows or cares how their phone calls get routed through the telephone network.

and self-protection modules. (*Early* versions of autonomic systems may be built in this way, however.) Instead, we expect these self-managing properties to be *emergent*, i.e. they will arise naturally from myriad interactions among individual constituents of the system that we refer to as *autonomic elements*. These *autonomic elements* will contain resources and deliver services to humans and to other autonomic elements. They will be responsible for managing themselves and their relationships with other elements. But the self-management of the system will be qualitatively more than the sum of the self-management of its parts—just as the social intelligence of an ant colony is qualitatively more than the sum total intelligence of the individual ants.

Autonomic elements will typically consist of one or more *functional units* coupled with a single *managerial unit* that controls and represents them. The managerial and functional units may or may not be physically co-located. The functional unit is essentially equivalent to what is found in ordinary non-autonomic systems, although it may have to be adapted to enable the managerial unit to monitor and control it. The functional unit could be a hardware resource, such as storage, CPU, or a printer. It could be a software resource, such as a database, a directory service, a service that converts among different file or message formats, or a large legacy system. At the very highest level, the functional unit could be an e-utility, an application service, or even an individual business. The managerial unit is what distinguishes the autonomic element from its non-autonomic counterpart, as it relieves humans of the responsibility of monitoring and managing the functional unit. The evolution towards fully autonomic computing is likely to proceed through gradual addition of increasingly sophisticated managerial units to existing functional units.

Each autonomic element will be responsible for managing its own internal state and behavior, and for managing its interactions with an environment that consists largely of signals and messages from other elements and the external world. Its internal behavior and its relationships with other elements will be driven by goals that are embedded within it by its designer, dictated to it by another element that has authority over it, or sub-contracted to it by a peer element with its tacit or explicit consent. The element may require assistance from other elements in order to achieve its goals or complete its tasks. If so, it will be responsible for obtaining the necessary resources from other elements, and for dealing with exception cases such as the unavailability or failure of a required resource.

Autonomic elements will be employed at many different levels, ranging from individual computing components such as individual disk drives to small-scale computing systems such as individual workstations or servers to entire automated enterprises that are situated in the largest autonomic system of all: the global economy. Especially at the highest levels of function and complexity, autonomic elements may have significant substructure beyond their division into managerial and functional units: they could be autonomic systems in their own right, with their own set of autonomic elements at the next level down. To take just one cross-section of such a layering as an example, consider an individual component like a disk drive, which might be part of a tightly coupled array of drives (RAID) that serves as part of a storage area network (SAN) used by a server farm that is part of an e-utility that provides web hosting services to computing systems of other companies for a fee. In some autonomic systems, this layering will look like a relatively strict hierarchy of management and control, with autonomic elements at higher levels dictating the behavior or the goals of those in the next level down. In other autonomic systems, there will still be identifiable levels of aggregation and boundaries of ownership, but the relationships among the autonomic elements will tend to be more biased towards peer-to-peer, with interactions that cut across several levels or boundaries.

At the lower levels of autonomic systems, an autonomic element's range of internal behaviors and the nature of its relationships with other elements may be relatively limited and hard-coded, and the set of elements with which it can interact may be relatively limited and hard-wired. This would tend to be most appropriate at the smaller scales of autonomic computing, down near the circuitry level, where pre-wired configurations of elements are essential for high performance, and there are few or no alternatives available for elements that fail or misbehave. Particularly at the level of individual components, well-established techniques—many of which fall under the rubric of fault-tolerance—have led to the development of elements that rarely fail, which is one important aspect of being autonomic. Decades of careful development and application of fault-tolerance techniques have produced marvels of engineering such as IBM zSeries servers, with a mean time to failure of several decades[13].

As one progresses to higher-level autonomic systems, fixed *behaviors*, *connections*, and *relationships* will evolve towards increased dynamism and flexibility. All of these aspects of autonomic elements will be expressed in more high-level, goal-oriented terms, leaving to the elements themselves the responsibility for resolving the details adaptively, on the fly. *Hardcoded behaviors* will give way to behaviors that are expressed as high-level objectives, such as "maximize this utility function", or "find a reputable message translation service", or "maintain this service level objective within 10% of its target value[4]." These high-level objectives will include business-level policies specified by humans, high-level goals encoded into autonomic elements by their designers, and ultimately high-level goals that are created automatically and dynamically by one element and passed along to another subordinate element. *Hardwired connections* among elements will give way to less and less direct specifications of an element's partners, from specification by physical address to specification by name and finally to specification by function, with the identity of the partner being resolved only at the moment that it is needed. *Hardwired relationships* will evolve into flexible and ephemeral relationships that are established via negotiation. Elements will handle new modes of failure, such as contract violation by a supplier, by restarting the process of finding and negotiating with new suppliers.

Autonomic elements will be both providers and consumers of services. Often, an element will serve in both roles, consuming services provided by other elements and combining and refining them into a new service. For example, a server element might rely upon one or more database elements for database services, while each database element could in turn rely upon one or more storage elements to help it satisfy its commitments to the server element. This leads naturally to an economic view of

[4]Under the right circumstances, control theory has been found to be a reasonable approach to maintaining service level objectives without depending upon a pre-existing model of the system [14].

an autonomic computing system as a self-assembling, adaptive web of interacting autonomic elements that draw upon one another's resources and services in an effort to satisfy their individual objectives. Indeed, economic principles appear likely to be an important resource allocation paradigm for autonomic systems.

In higher-level autonomic systems, *service-oriented* architectural concepts such as are embodied in Web Services and Grid Services languages and technologies [15, 16] will play an important role in autonomic computing. However, service-oriented architecture will not by itself provide a sufficient foundation for autonomic computing. In their role as service providers, autonomic elements will not unquestioningly honor requests for service, as would typical objects invoked in an object-oriented environment, or typical software components, or today's Web Services. They will only provide a service if to do so is consistent with their goals. Moreover, in their role as consumers, autonomic elements will autonomously and proactively issue requests to other elements in order to carry out their objectives. They will have complex life cycles, continually carrying on multiple threads of activity, continually sensing and responding to the environment in which they are situated. Autonomy, proactivity, and goal-directed interactivity with their environment are distinguishing characteristics of software agents. Thus, the higher levels of autonomic computing systems will be *multi-agent systems* populated with large numbers of software agents functioning as autonomic elements, and *agent-oriented* architectural concepts will be critically important in the design of autonomic computing systems [17].

4 Engineering Challenges

In this section, we discuss some of the most significant engineering challenges that will be encountered in designing and implementing autonomic systems. The challenges faced at the lower, more hardwired levels will tend to be either more familiar (such as error discovery techniques on memory chips or instruction re-try on IBM zSeries CPU chips [18]), or else they will be a subset of those occuring at the higher, more dynamic levels. Therefore, to expose as many new issues as possible, we shall focus mainly on the higher, more distributed levels of autonomic computing. We shall progress from issues pertaining to the individual autonomic element itself to those concerning relationships among autonomic elements, culminating in a discussion of system-wide issues and interfaces between humans and autonomic systems.

4.1 Life cycle of an autonomic element

As a vehicle for exploring some of these challenges inherent in developing and deploying an autonomic element, we shall consider its life cycle, beginning with how it is programmed, continuing with testing and verification, on to installation, configuration, optimization, upgrading, monitoring, problem determination and recovery, and culminating finally in de-installation or replacement.

Consider first the challenge of *programming* autonomic elements. The programming model is likely to be agent-oriented—that is, it will entail interactions among "autonomous components (agents) that have particular objectives to achieve[17]." Major issues include how to represent high-level tasks and capabilities, and how to write planning and execution engines that map these high-level specifications into lower-level actions. Programming an autonomic element will require encoding within it *policies*: high-level specifications of goals or constraints, typically in the form of rules or utility functions. It may also entail coding within the element the means to acquire policies on the fly from administrators or other elements. Programmers will also need to be concerned with how an element will negotiate and otherwise interact (perhaps strategically) with potential customers and suppliers.

Testing autonomic elements and *verifying* that they behave correctly will be especially challenging in large-scale systems, since in general it will be difficult to anticipate the environment in which the elements will be situated—especially when the environment extends across multiple administrative domains or enterprises. Testing networked applications that require coordinated interactions among several autonomic elements will be even more difficult. Designers and owners of autonomic elements might gain some measure of comfort by placing them within simulation or testbed environments that automatically exercise them in various ways. However, such techniques will never be able to *verify* that an autonomic element (or a set of autonomic elements) will behave as intended under all circumstances. It will be virtually impossible to build test systems that capture the size and complexity of the actual systems they are simulating, and moreover it will be virtually impossible to generate and run through the simulator a set of workloads that will capture all of the possible event sequences that may occur in the real system. It might be possible to test newly deployed autonomic elements *in situ* by having them perform alongside more established and trusted elements of similar functionality. Potential customers of a given element may also care to test and verify its behavior, both prior to the establishment of a service agreement, and during the provision of that service (to ensure that it is being delivered as promised). One approach is for the autonomic element to attach a testing method to its service description.

Installation and *configuration* of autonomic elements will most likely entail a bootstrapping process, beginning with the element registering itself (publishing its capabilities and contact information) to a directory service. The element might also use the directory service to discover suppliers or brokers that may provide information or services needed to complete its initial configuration, and to seek out potential customers or brokers to which it can delegate the task of finding customers.

Monitoring will be an essential feature of autonomic elements. Elements will continually monitor themselves to ensure that they are meeting their own objectives, and they will log this information to serve as the basis for adaptation, self-optimization, or reconfiguration. Autonomic elements will also continually monitor the performance of their suppliers, to ensure that they are receiving the agreed-upon level of service, and they will monitor their customers, to ensure that they are not exceeding the agreed-upon level of demand. There may even be special sentinel elements[5] whose purpose is solely to monitor the behavior of other elements, and alert other elements

[5]Sentinels are a familiar and powerful concept from the multi-agent systems literature.

when they fail.

Monitoring will also be important because, especially when coupled with event correlation and other forms of analysis, it supports *problem determination* and *recovery* when a fault is found or suspected. It will be a challenge to apply monitoring, audit and verification tests at all the points that they are needed, without burdening systems with excessive bandwidth or processing demands. Technologies to allow statistical or sample-based testing in a dynamic environment may prove helpful.

Envisioning autonomic systems as a complex supply web, it seems apparent that some aspects of *problem determination* will become easier than they are currently, while others will become harder. An autonomic element that detects poor performance or failure in a supplier may not care about the reason—it may simply work around the problem by finding a new supplier to fulfill its needs. In other situations, however, it will be necessary to determine why one or more elements are failing, and it will be necessary to do so without shutting down and restarting the entire system. Here the complex and ever-changing connectivity of the supply web will hamper efforts to trace problems to their source. We will need theoretically-grounded tools for tracing, simulation, and problem determination in complex dynamic environments.

Particularly when autonomic elements (or applications based on interactions among multiple elements) have a large amount of state, it will be challenging to *recover* gracefully and quickly from failure, or to restart applications after software has been upgraded, or after function has been relocated to new machines. Today, transaction systems, transactional applications, and database systems require the state to be reconstructed by loading and initializing large programs and reading huge log files—a process that can take half an hour or more. We must find ways to reduce the restart or recovery time by at least two orders of magnitude, and to avoid reconstructing the portion of the state that contained or led to the original fault. Researchers at the University of California at Berkeley and Stanford University have made a promising start in this direction[19, 20].

Autonomic elements will need to *upgrade* themselves from time to time. They might subscribe to a service that alerts them to the availability of relevant upgrades, and decide for themselves when to apply the upgrade (possibly with guidance from another element or a human). Alternatively, entirely new elements incorporating the upgrade could be created afresh as part of a system upgrade, and outmoded elements could be eliminated only after the new ones establish that they are working properly.

We have highlighted a few of the many challenges faced by autonomic elements at several of the stages in their life cycle. An additional challenge is the overall management of the life cycle of an autonomic element. Autonomic elements will typically be engaged in many activities simultaneously—participating in one or more negotiations that are at various phases of completion, proactively seeking inputs from other elements, etc. They will need to schedule and prioritize the myriad activities in which they are involved. They will need to represent their life cycle, both to reason about it and to communicate it to other elements. Particularly long-lived autonomic elements may outlive their own hardware. In such cases, the life cycle management must migrate an element to other hardware prior to any hardware upgrade, and it may need to migrate the element to the new hardware once it is in place.

4.2 Relationships among autonomic elements

In its most dynamic and elaborate form, the service relationship between two or more autonomic elements will also have a life cycle. Elements will *specify* their capabilities and their needs, *locate* one another, and *negotiate* with one another to establish an agreement. The elements that are parties to the agreement will then execute the established agreement. First, they will *provision* their resources to allow the service(s) to be provided; this may require a provider to procure services to help it meet its obligation. Then, they will *operate* under the negotiated agreement, and finally they will *terminate* the relationship. Each stage of this life cycle engenders its own set of engineering challenges and standardization requirements.

1. **Specify**. An autonomic element must have associated with it a set of output services it can perform and/or a set of input services that it requires, expressed in a standard format so that it can be understood by other autonomic elements. Typically, the element will register a description of its capabilities, and details about addresses and protocols that can be used to communicate with it, with a directory service such as Universal Description, Discovery, and Integration (UDDI) [21] or the Open Grid Services Architecture (OGSA) Registry [16]. Establishing standard service ontologies and a standard service description syntax and semantics that is sufficiently expressive for machines to interpret and reason about is an area of active research, of which the the Semantic Web effort [22] sponsored by DARPA is representative.

2. **Locate**. An autonomic element must locate input services that it needs, and must be located by other elements that want to use its output services. If it has a simple, static relationship with other elements, this just amounts to following a pointer. If it must locate other elements dynamically, it may look them up by name or by function in a directory service, possibly by means of a search process that involves sophisticated reasoning about service ontologies. Then, the element may contact one or more potential service providers directly and engage in a conversation or a negotiation to determine whether they can obtain exactly the service they require. Autonomic elements will rely upon naming facilities provided by their environment, and secure mechanisms for establishing one another's identities.

   In general, but especially when the system spans multiple trust boundaries, autonomic elements will need to choose potential partners on the basis of their suitability or reliability. They will base such decisions on their own experience or on information or recommendations from third-party reputation services. While automated reputation and automated recommendation techniques have been the subject of some research for the last half dozen years or more [23, 24], several significant challenges remain. For example, the electronic environment affords numerous opportunities for cheating and collusion, which require more robust reputation or recommendation mechanisms than have yet been proposed.

3. **Negotiate**. Once an element finds potential providers of an input service, it

must negotiate with them in order to obtain that service. We construe "negotiation" very broadly to be any process by which an agreement is reached. In situations in which the element providing a service is subservient to the one requesting it, the original request will really be a demand for service, and that request will constitute the agreement unless the provider is unable to satisfy it due to resource limitations. Other simple forms of negotiation are *first-come, first-served*, i.e. the provider satisfies all requests until it runs into resource limitations, and *posted price*, i.e. the provider sets a price (in real or artificial currency) for its service, and the requester must take it or leave it. More complex forms of negotiation range from various forms of *bilateral negotiation* involving proposals and/or counterproposals, and *multilateral negotiations* over multiple attributes, such as price, service level, priority, etc. Especially when they are multi-lateral, negotiations may be assisted by a third-party arbiter.

Several aspects of negotiation will be rich sources of engineering and scientific challenges for autonomic computing. First, elements require flexible ways to express multi-attribute needs and capabilities that go beyond simple utility functions and constraints, and they need mechanisms for deriving these expressions of value or cost from human input (e.g., via preference elicitation) or from computation. Second, elements will need effective negotiation strategies that satisfy as optimally as possible the individual goals of the elements, or the goals of the system as a whole. Third, there is a need for protocols that establish the rules of negotiation and govern the flow of messages among the negotiators. A promising approach is for each element to execute its local copy of a shared conversation policy [25, 26] that specifies the set of messages that can be transmitted or received in any given conversational state, and the state transitions that occur upon the transmission or receipt of each message. Finally, there are several challenges associated with service agreements, the culminations of successful negotiations. There is a need for languages that express service agreements in their transient and final forms. This entails standardizing the syntax and semantics for describing the various service attributes (such as duration, cost, latency, and bandwidth), the procedures to be used for detecting failures and resolving disputes, and other details. Efforts to standardize the representation of agreements, such as tpaML (Trading Partner Agreement Markup Language) [27], are being developed under the ebXML initiative. There is a need for automated mechanisms for negotiating [28], enforcing, and reasoning about agreements; early work in the agents community may serve as a good foundation. Two additional challenges that remain largely unaddressed include mechanisms for translating terms of agreements into plans of action and solutions to resource deadlock—a classical problem that will take on new guises.

4. **Provision**. Once an agreement has been reached, the parties to the agreement must provision their internal resources so that the service can be provided. This may be as simple as noting in an access list that a particular element may request service in the future, or it may entail the establishment of additional relationships with other elements to which part of the agreed-upon service or

task is subcontracted.

5. **Operate**. Once both sides are properly provisioned, they may operate under the negotiated agreement. The managerial unit of the service provider would oversee the operation of its functional unit, monitoring it to ensure that the agreement is being honored; the service requester might similarly monitor the level of service. If the agreement is not being met, one or both of the elements would seek an appropriate remedy—including assessing a penalty, renegotiating the agreement, and taking technical measures to minimize any harm from the failure. If the problem were to persist, the service requester might terminate the agreement and renegotiate a new one with another party.

6. **Terminate**. When the agreement has run its course, the parties agree to terminate it, possibly freeing their internal resources for other uses and terminating agreements for input services that are no longer needed. Pertinent information about the service relationship with the various partners may be recorded in a reputation database.

4.3 System-wide issues

There are some additional, very important engineering issues that arise at the level of the system as a whole. Among these are security, privacy, and trust, and the emergence of new types of services that are not extensions of traditional functional units, but are created to serve the needs of other autonomic elements.

Autonomic computing systems will be subject to essentially all of the issues in security, privacy and trust that exist in traditional computing systems. Autonomic elements and systems will need to both establish and abide by security policies, just as human administrators do today, and they will need to do so in a manner that is understandable and failsafe. Systems that span multiple administrative domains—and especially those that cross company boundaries—will be subject to many of the challenges that now confront electronic commerce, including authentication, authorization, encryption, signing, secure auditing and monitoring, non-repudiation, data aggregation and anonymization, and compliance with complex legal requirements that vary from state to state or country to country.

The autonomic systems infrastructure must provide autonomic elements with the means to reliably identify themselves, reliably verify the identities of other entities with which they communicate, reliably verify that a message has not been altered in transit, and ensure that messages and other data are not read by unauthorized parties. Elements must also appropriately protect private and personal information that comes into their possession, to satisfy privacy policies and privacy laws. Privacy measures that keep data segregated according to its origin or its purpose must be extended into the realm of autonomic elements to satisfy policy and legal requirements.

Elements of autonomic systems, and the infrastructure upon which they are built, must be robust against attacks. Autonomic systems may be prone to new and insidious forms of attack. In traditional computing systems, attackers alter a system's behavior via direct intervention. In autonomic computing systems, an attacker might attain much greater leverage by altering either the system's goals or its monitors. Preventing problems of this sort may require a new subfield of computer security that seeks to thwart fraud and fraudulent persuasion of autonomic elements.

At the larger scales, autonomic elements will be agents, and autonomic systems will be multi-agent systems. Accordingly, they will possess standard facilities, most likely based on Web Services or OGSA infrastructures, that support naming, locating and communicating among agents, as well as means for managing the creation, destruction, or admission of new agents. They will include facilities that enable agents to communicate with agents on other platforms [29], and permit migration of agents across platforms. Autonomic systems will be inhabited by "middle agents" [30] that serve as intermediaries of various types, including directory services, matchmakers, brokers, auctioneers, data aggregators, dependency managers (for detecting, recording and publicizing information about functional dependencies among autonomic elements), event correlators, security analysts, timestampers, and sentinels and other types of monitors that assess the health of other elements, or the system as a whole. Traditionally, many of these services have been part of the system infrastructure; in a multi-agent, autonomic world it will be more natural and flexible to move them out of the infrastructure and represent them as autonomic elements (or agents) themselves.

4.4 The Human-Computer Interface (HCI)

While autonomic systems will assume much of the burden of system operation and integration, it will still be up to humans to provide autonomic systems or autonomic elements with policies—the goals and constraints that will govern their actions. The enormous leverage of autonomic systems will greatly reduce the number of errors made by humans. On the other hand, this leverage will greatly magnify the consequences of any errors that humans make in specifying the goals. Furthermore, the indirect effect of policies on system configuration and behavior will make policy errors very difficult to trace and correct. Therefore it will be critically important to ensure that the specified goals represent what is really desired. This defines two related sets of engineering challenges: ensuring that goals are specified correctly in the first place, and ensuring that systems behave reasonably even when they are not.

First, consider the problem of ensuring that goals are specified correctly. In many cases, the set of goals will be complex, multi-dimensional, and conflicting. Even a goal as superficially simple as "maximize utility" will often require a human to express a complicated multi-attribute utility function. A key to reducing error will be to simplify and clarify the means by which humans express their goals to computers. Psychologists and computer scientists will need to work together to define new languages, metaphors, and technologies for expressing goals and for visualizing or simulating their likely effect. The right balance will have to be struck between overwhelming humans with too many questions or too much information and under-empowering them with too few options or too little information.

Second, consider the problem of ensuring reasonable system behavior in the face of erroneous input. This is just another facet of robustness: autonomic systems will need to protect themselves from input goals that are inconsistent, implausible, dangerous, or unrealizable with the resources at hand. Autonomic systems will subject such inputs to extra validation, and when self-protective measures fail they will rely on deep-seated notions of what constitutes acceptable behavior in order to detect and correct problems. In some cases (such as resource overload) they will inform human operators about the nature of the problem and offer alternative solutions.

Even in highly autonomic systems, human administrators will need interfaces for monitoring and manually controlling autonomic computing systems and their components. By visualizing behavior and behavioral specifications at all levels, administrators can verify that systems are operating as intended. Such interfaces will help build trust in autonomic systems, and they will permit administrators to intervene on those increasingly rare occasions when automated procedures prove inadequate. Integrating occasional low-level interventions smoothly with high-level policy specifications will be challenging, but less and less necessary.

5 Scientific Challenges

While much of the burden of constructing and designing autonomic systems will fall to systems and software architects, our ultimate success will hinge on the extent to which theorists can identify universal principles that span the multiple levels at which autonomic systems can exist, from systems to enterprises to economies. In this section, we enumerate a few basic theoretical issues, the investigation of which may lead to the discovery of universal characteristics of autonomic systems, and universal principles for designing and controlling them.

A scientific challenge that lies at the heart of autonomic computing is to define appropriate abstractions and models for understanding, controlling, and designing emergent behavior in autonomic systems. There is a need for fundamental mathematical work aimed at understanding how the autonomic properties of self-configuration, self-optimization, and self-maintenance and robustness, and related issues such as the stability (or nonlinear dynamics), predictability, and performance of the system and its constituent elements arise from or depend upon

- the behaviors, goals, and degree of adaptivity of the individual autonomic elements,
- the pattern and type of interactions among them, and
- the external influences experienced by (or demands placed upon) the system.

Understanding the mapping from local behavior to global behavior is a necessary but insufficient condition for being able to *control* and *design* autonomic systems. We must also figure out how to exploit the inverse relationship: How can we derive a set of behavioral rules and interaction rules and patterns that, if imbedded within individual autonomic elements, will induce a desired global behavior? The nonlinearity of emergent behavior makes such an inversion highly non-trivial. One plausible approach couples advanced search and optimization techniques with parameterized models of the local-to-global relationship and the likely set of environmental influences to which the system will be subjected. This approach has been pioneered by Mitchell et al. at the Santa Fe Institute, who have used genetic algorithms to evolve the local transformation rules of simple cellular automata to achieve desired global behaviors [31]. Wolpert and colleagues have developed an alternative for cooperative multi-agent systems called the COIN (COllective INtelligence) framework [32]. Given a high-level global objective, they derive individual goals for the agents which, when followed selfishly by each agent, result in the desired global behavior. But these efforts are just a start. It remains a challenge to understand the fundamental limits on what classes of global behavior can be achieved, and to develop truly practical methods for designing the emergent behavior of systems to a given specification.

The methods of Mitchell, Wolpert and their colleagues are aimed at establishing the rules of a system at design time. But autonomic systems must deal with continual shifts in demand, and changes in their own structure and function that can only be known at run time. Control theoretic approaches may prove very useful in this capacity. In particular, managerial units may use control systems to govern the behavior of their associated functional units. The greatest value is likely to be found in extensions to distributed or hierarchical control theory, which consider interactions among independently or hierarchically controlled elements, rather than focussing purely on an individual controlled element. Newer paradigms for control may be needed in cases where there is not a clear separation of scope or time scale.

A second, related scientific challenge is to develop a theory of robustness for autonomic systems, including definitions and analyses of robustness, diversity and redundancy, and their relationship to one another. While there has been some interesting work on tradeoffs between robustness and optimality over the last several years, much remains to be done. It is encouraging to see that the Santa Fe Institute is nucleating a multidisciplinary study of robustness[33].

A third fundamental scientific challenge is to develop a theoretical foundation for learning and optimization in both cooperative and competitive multi-agent systems. There is a great need for results that cover both the perspective of individual agents (i.e. what learning and optimization techniques are best suited to multi-agent environments) and the perspective of the system as a whole (i.e. what are the possible collective modes of behavior that may exist in a society of adaptive agents). Machine learning by a single agent in relatively static environments is well studied, and well supported by strong theoretical results. However, in the more sophisticated autonomic systems, individual elements will be agents that continually adapt to their environment—an environment that consists largely of other agents. Thus, even if external conditions do not vary, the fact that individual agents are adapting to other adaptive agents violates the traditional assumptions on which single agent learning theories are based. One can try to apply traditional machine learning techniques despite the lack of theoretical guarantees on convergence; in some cases this happens to work, and in others it leads to interesting forms of instability [34]. Learning in multi-agent systems is a very challenging problem, but it is relatively unexplored. There are virtually no major theorems, and only a small number of empirical results. The few authors who have ventured into learning in multi-agent systems have tended to combine methods of game theory with machine learning, but this approach may not be practical for nontrivial problems with moderate to large numbers of agents.

Just as learning becomes a much more interesting and challenging problem in multi-agent systems, so does optimization. The root cause is the same—whether it is because they are learning or because they are optimizing, agents are changing their behavior, making it necessary for other agents to change *their* behavior, and so on, potentially leading to instabilities. Optimization in such an environment will need to deal with dynamics that may be created by a collective mode of oscillation rather than a drifting environmental signal. Optimization techniques that inherently assume a stationary environment have been observed to fail pathologically in multi-agent systems in various ways [35, 36], and will therefore have to be either revamped or replaced with new, inherently dynamic optimization methods.

A fourth general scientific challenge for autonomic computing is to establish a solid theoretical foundation for negotiation from two perspectives: that of the individual autonomic elements, and that of the system as a whole. From the perspective of individual elements, it is important to develop and analyze algorithms and negotiation protocols, and to determine what bidding or negotiation algorithms are most effective. At the system level, it is important to establish how the overall system behavior depends upon the mixture of negotiation algorithms employed by the population of autonomic elements, and to establish the conditions under which multilateral (as opposed to bilateral) negotiations among elements are necessary and/or desirable.

A fifth general scientific challenge is to automate to the fullest possible extent the construction of statistical models of large networked systems that allow overall performance problems to be detected or predicted from a stream of sensor data from individual devices. At long time scales (roughly the scale at which the configuration of the system changes), we seek methods that automate the intelligent aggregation of statistical variables to reduce the dimensionality of the problem to a size that is amenable to adaptive learning and optimization techniques that operate on shorter time scales.

A sixth and final scientific challenge is to develop appropriate theories and theoretical constructs for measuring, understanding, and proving properties of autonomic systems. These may include

- a process algebra with general primitives for initiating, monitoring, moving, killing, retrying, restarting, compensating autonomic elements,
- methods for guaranteeing idempotency of operations, and
- representation of tasks and services, with composition/decomposition rules, conflict graphs, and a general algebra and logic of tasks.

These and numerous other fundamental questions will require a concerted effort by researchers in a diverse set of fields, including the science of complexity and emergent phenomena, machine learning, economics, and computer science.

6 Conclusions

We believe that the grand challenge of creating autonomic computing systems can be met. Realizing our long-term vision won't require magic, and it won't require full solution of the AI problem, but it will require a worldwide, multidisciplinary effort by researchers in academia and industry to address several deeply challenging scientific and engineering problems, some of which have been outlined in this article. Long before many of the more challenging problems are solved, less automated realizations of autonomic systems will prove to be extremely valuable[6], and their value will increase substantially as autonomic computing technology improves and earns greater trust and acceptance.

Many subfields of computer science will be called upon as we reach towards the long-term vision of autonomic computing, including software architecture, programming languages, agents, machine learning, and many others. But now that computing technology is so deeply woven into the fabric of our daily lives, it is fitting that this latest major paradigm shift in computing must draw upon disciplines that lie far beyond the traditional boundaries of computer science. We will look to scientists studying nonlinear dynamics and complexity for new concepts and theories of emergent phenomena and robustness, to economists and e-commerce researchers for ideas and technologies regarding negotiation and supply webs, and to psychologists and human factors researchers for new goal-definition and visualization paradigms, and for ways to help humans build trust in autonomic systems. We anticipate contributions from the legal profession as well, as many of the same issues that are arising in the context of electronic commerce will be important in autonomic systems that span organizational or national boundaries. How can we get autonomic elements to understand and adhere to laws and regulations? Can autonomic elements enter into legally binding agreements with one another? Who is legally responsible for their decisions and actions?

It will be a challenge in itself to bridge the language and cultural divides among the many different disciplines that will be brought together in this endeavor, and to harness this diversity to yield successful and perhaps universal approaches to autonomic computing. It will be interesting to see what new cross-disciplines develop as we begin to work together to solve the fundamental problems of autonomic computing.

Acknowledgments

The author is indebted to the many people who influenced this paper with their ideas and thoughtful criticisms. Special thanks go to David Chess and David Chambliss for contributing and summarizing valuable thoughts on life cycle, security, and HCI issues. The author is also grateful to Bill Arnold, David Bantz, Rob Barrett, Peter

[6]One good first step is to introduce better instrumentation into today's systems, so that their state can be understood in greater detail. Even before we invent mechanisms that can effectively exploit detailed information about system state, such information will benefit system administrators greatly.

Capek, Alan Ganek, Germán Goldszmidt, James Hanson, Joseph Hellerstein, James Kozloski, Charles Peck, Ed Snible, and Ian Whalley for their helpful comments, and to members of the IBM Academy team for their extensive written and verbal contributions to this document. Members of the IBM Academy effort included Lisa Spainhower and Kazuo Iwano (co-leaders), William H. Tetzlaff (technical committee contact), Robert Abrams, Sam Adams, Steve Burbeck, David Chambliss, Bill Chung, Denise Y. Dyko, Stuart Feldman, Lorraine Herger, Mark Johnson, James Kaufman, David Kra, Ed Lassettre, Andreas Maier, Timothy Marchini, Norm Pass, Colin Powell, Stephen A. Smithers, Daniel Sturman, Mark N. Wegman, Steve R. White, and Daniel Yellin.

References

[1] IBM. Autonomic computing: IBM's perspective on the state of information technology. Available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[2] Paul Horn. Keynote speech to National Academy of Engineers. Delivered at Harvard's Division of Engineering and Applied Sciences, March 8, 2001.

[3] Richard Morochove. Big Blue bets big on little Linux. *Toronto Star*, August 24, 2000.

[4] Harris Miller. Despite decline in IT workforce, the numbers indicate future optimism. URL: http://www.itaa.org/news/view/ViewPoint.cfm?ID=23, May 31, 2002.

[5] Jeffrey O. Kephart, Gregory B. Sorkin, and Morton Swimmer. An immune system for cyberspace. In *Proceedings of the IEEE Symposium on Systems, Man, and Cybernetics*, pages 879–884, 1997.

[6] Steven Hofmeyr and Stephanie Forrest. Architecture for an artificial immune system. *Evolutionary Computation Journal*, 7(1):45–68, 2000.

[7] Marco Dorigo and Gianni Di Caro. Ant colony optimization: A new meta-heuristic. In *1999 Congress on Evolutionary Computation*, pages 1470–1477, Piscataway, NJ, 1999. IEEE Service Center.

[8] Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[9] Richard J. Niemiec, Joe Trezzo, Rich Niemiec, and Bradley D. Brown. *Oracle Performance Tuning Tips and Techniques.* McGraw-Hill Professional Publishing, 1999.

[10] Guy Harrison. *Oracle SQL High-Performance Tuning.* Prentice Hall, 2000.

[11] Jonathan Lewis. *Practical Oracle 8i: Building Efficient Databases.* Addison-Wesley, 2001.

[12] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, Denver, August 2000.

[13] IBM eServer zSeries e-commerce solutions. http://www-1.ibm.com/servers/solutions/ecommerce/zseries/.

[14] S. Parekh, N. Gandhi, Joseph L. Hellerstein, Dawn Tilbury, T. S. Jayram, and Joseph Bigus. Using control theory to achieve service level objectives in performance management. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[15] Heather Kreger. Web services conceptual architecture (WSCA 1.0). Available at http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf, 2001.

[16] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: Open grid services architecture for distributed systems integration. Draft, February 2002.

[17] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.

[18] IBM. Introducing the IBM eServer zSeries 900 platform: A self-managing, multi-system server. Available at http://www-1.ibm.com/servers/eserver/zseries/library/whitepapers/pdf/gf225174.pdf, 2000.

[19] G. Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[20] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report CSD-02-1175, University of California at Berkeley CS Department, March 2002.

[21] Ariba, IBM, and Microsoft. UDDI technical white paper. http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, 2000.

[22] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[23] G. Zacharia, A. Moukas, and P. Maes. Collaborative reputation mechanisms in electronic marketplaces. In *32nd Hawaii International Conference on System Sciences*, 1999.

[24] Bin Yu and Munindar P. Singh. A social mechanism of reputation management in electronic communities. In *Cooperative Information Agents*, pages 154–165, 2000.

[25] Mark Greaves, Heather Holmback, and Jeffrey Bradshaw. What is a conversation policy? In *Issues in Agent Communication*, pages 118–131, 2000.

[26] James E. Hanson, Prabir Nandi, and David W. Levine. Conversation-enabled web services for agents and e-business. In *Proc. 3rd International Conference on Internet Computing (IC 2002)*, 2002.

[27] A. Dan, D. M. Dias, R. Kearney, T. C. Lau, T. N. Nguyen, F. N. Parr, M. W. Sachs, and H. H. Shaikh. Business-to-business integration with tpaML and a business-to-business protocol framework. *IBM Systems Journal*, 40(1), 2001.

[28] N. R. Jennings, P. Faratin, A. R. Lumuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated negotiation: Prospects, methods and challenges. *International Journal of Group Decision and Negotiation*, 10:199–215, 2001.

[29] Foundation for Intelligent Physical Agents web page. URL: http://www.fipa.org.

[30] H. Wong and K. Sycara. A taxonomy of middle-agents for the internet. In *Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS-2000*. IEEE Computer Society, 2000.

[31] Rajarshi Das, James P. Crutchfield, Melanie Mitchell, and James E. Hanson. Evolving globally synchronized cellular automata. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 336–343, San Francisco, CA, 1995. Morgan Kaufmann.

[32] David Wolpert, Kevin Wheeler, and Kagan Tumer. Collective intelligence for control of distributed dynamical systems. Technical Report NASA-ARC-IC-99-44, NASA, 1999.

[33] Santa Fe Institute robustness program web page. http://discuss.santafe.edu/robustness.

[34] Jeffrey O. Kephart and Gerald J. Tesauro. Pseudo-convergent Q-learning by competitive pricebots. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML'00)*, pages 463–470, Stanford, CA., 2000.

[35] Jeffrey O. Kephart, Rajarshi Das, and Jeffrey K. MacKie-Mason. Two-sided learning in an agent economy for information bundles. In *Agent Mediated Electronic Commerce*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000.

[36] Jeffrey O. Kephart, Christopher H. Brooks, Rajarshi Das, Jeffrey K. MacKie-Mason, Robert S. Gazzale, and Edmund H. Durfee. Pricing information bundles in a dynamic environment. In *Proceedings of ACM EC-01*, 2001.

Magic

Dr. David West
New Mexico Highlands University

"Any sufficiently advanced technology is indistinguishable from magic"
Arthur C. Clarke
Profiles of the Future

For the purposes of this essay we will assume an obverse version of Clarke's insight. "If a technology is not magical it is insufficiently advanced." Computing and software development are clearly not magical even though some applications, especially in cinema special effects, certainly convey magical impressions.

The central question of this essay – can we use magic as a metaphor to re-evaluate and redefine the theory and practice of computing? Or, stated slightly differently, can magic provide a metaphor for opening a new frontier in the investigation and solution of the core problems confronted by software developers and computing professionals in today's world?

Along the way to answering this question we will explore, for just a bit, the appeal of the metaphor and make two side trips to examine potential insights from other disciplines.

The Appeal of Magic

Magic is not a new metaphor for computing. Adept practitioners have long been referred to as 'wizards." In 1981, Vernor Vinge published a novella, *True Names*, using magic as a key metaphor. Twenty years later, Vinge wrote about why magic seemed so appropriate for his fictional account of networked computing in a place he called "The Other Plane" but which today most people call cyberspace.

> *"…. Even in serious commercial programming, the magic metaphors are very common, partly as humor, partly because they provide useful terminology to hang reasoning on. … So the magical terminology fit with some things that go on in real programming. … The magic metaphor was a powerful guide in the choosing of terms …"* (Frenkel 2001, pp 18-20)

Vinge's work resonated with numerous computer scientists, both at the time it was first published and today. Marvin Minsky wrote an afterword that accompanied the first publication and he was joined by the likes of Pattie Maes, Danny Hillis, and Richard Stallman in writing essays about the influence of *True Names* in the world of computing.

The use of magical metaphor in True Names is mostly at the human-computer interface.

> *"Protected now against traceback, Mr. Slippery set out for the Coven itself. He quickly picked up the trail, but this was never an easy trip, for the SIG members had no interest in being bothered by the unskilled.*

> *In particular, the traveler must be able to take advantage of subtle sensor indications, and see in them the environment originally imagined by the SIG. The correct path had the aspect of a narrow row of stones cutting through a gray-greenish swamp. The air was cold but very moist. Weird, towering plants dripped audibly onto the faintly iridescent water and the broad lilies. The subconscious knew what the stones represented, handled the chaining of routines from one information net to another, but it was the conscious mind of the skilled traveler that must make the decisions that could lead to the gates of the Coven, or the symbolic 'death' of a dump back to the real world.*
>
> *There was much misinformation and misunderstanding about the Portals. Oh, responsible databases like the LA Times and the CBS News made it clear that there was nothing supernatural about them or the Other Plane, that the magical jargon was at best a romantic convenience and at worst obscurantism.*
>
> *A typical Portal link was around fifty thousand baud, far narrower than even a flat video channel. Mr. Slippery could feel the damp seeping through his leather boots, could feel the sweat starting on his skin even in the cold air, but this was the response of Mr. Slippery's imagination and subconscious to the cues that were actually being presented through the Portal's electrodes. The interpretation could not be arbitrary or he would be dumped back to reality and would never find the Coven; to the traveler on the Other Plane, the detail was there as long as the cues were there. And there is nothing new about this situation. Even a poor writer - if he has a sympathetic reader and an engaging plot - can evoke complete internal imagery with a few dozen words of description. The difference now is that the imagery has interactive significance, just as sensations in the real world do. Ultimately, the magic jargon was perhaps the closest fit in the vocabulary of millennium Man." (Frenkel 2001, pp. 251-252)*

Although interface issues are critical, and although HCI designers have yet to apply all the insights available in True Names and subsequent science fiction works like William Gibson's *Neuromancer* and Neal Stephenson's *Diamond Age*; this essay is more concerned with looking at magic as a metaphor to redefine the very essence of computing itself.

To accomplish the main goal we need a better understanding of what is meant by magic and what referents we want to associate with the metaphor when we apply it to the world of computing. Then we need a basis (a theory or at least an ideational framework) for applying the metaphor. For the first we will turn briefly to anthropological framings of magic and for the second a superficial examination of one aspect of Hindu philosophy.

Magical Essentials

A belief in the supernatural is a cultural universal - all cultures of which we are aware, even prehistoric ones exhibit some kind of belief in forces or spirits that transcend the material world. It is useful to make a relatively clear distinction between a belief in "forces" and in "spirits." Spirits usually have qualities like bodily form, personality, predictable responses to human beings, etc.

Supernatural forces, on the other hand, usually have no will of their own and cannot refuse humans who know how to invoke, command, and manipulate them. Forces, can be used for 'good' or for 'bad,' by humans who know the proper rites and spells - i.e. who know magic.

The popular conception of magic incorporates both spirits and forces. Readers of J. K. Rowlings popular "Harry Potter" novels confront spells that affect forces and inanimate objects as well as a collection of supernatural creatures ranging from house elves to 'veela.' As entertaining as such things might be, we will confine ourselves to the realm of forces and leave Hagrid to be responsible for the care of magical creatures.

In 1890, Sir James Frazer proposed two logical principles or assumptions common to all forms of magic: the imitative principle (like causes like), and the contagious principle (contact based). A "voodoo" doll is an example of imitative magic. Spells performed on hairs, nail clippings, jewelry, associated with the target of the spell are examples of contagious magic.

Rituals are organized performances of behaviors intended to influence or manipulate supernatural forces. Rituals are stereotyped - the same behaviors in the same order, the same speech patterns, the same places, the same language, the same objects of magical manipulation. Rituals may be as simple as a single word (uttered in exactly the correct way and at the correct time in the correct place, addressed to the correct object); or, they may involve a cast of thousands, be extra-ordinarily complex and take many days to complete. Rituals range in their intended outcome from manipulations intended to invoke an immediate result, to those that have no direct result, but merely re-establish balances or harmonies among natural and supernatural forces.

From this exceptionally brief examination of the anthropological notion of magic we extract the following ideas that we will apply to computing later in the essay.

- Users of magical things invoke responses that are intrinsic to the magical object, they do not concern themselves with the nature of the force (what the Polynesians called *mana*) that enables the magical thing to respond. (Reminiscent of objects and black-box design, but stricter in its application - there is no "inside" of a magical thing like there is of a black-box or object.)
- All interaction between users and magical objects is of the form, stimulus-response. The stimulus is the ritual, the response the action of the invoked force (hopefully the desired outcome of the invocation). If the ritual was faulty there is no response or an undesired response. (Superficially similar to message passing, if messages are restricted to unary imperatives.)
- User interface design will conform to limitations derived from the two principles of magical invocation - contagion and imitation.
- Magical objects are limited to a specific set of responses. You must find the correct spirit and use the correct incantation if you want a result. But, this limitation is also, at least partially, a benefit - providing a way to categorize ritual-object-response triads in order to form 'indices' that will allow us to find the one we need. (Again, some surface similarity to ideas of classes and class hierarchies but only in terms of taxonomic organization and classification - not inheritance.)

Metaphysics for Magicians

Vedic (Hindu) philosophers posited an ethereal dualism - separate realms of pure "mind" (*purusa*) and pure inert matter (*prakrti*). Some kind of cosmic accident caused the two realms to infuse one another giving rise to the phenomenological universe of which we find ourselves. Fundamental tenets of Hinduism and Buddhism - reincarnation, enlightenment, Karma, among others - are grounded in this basic metaphysics.

A corollary of this philosophy is the assertion that every bit of *prakrti* - from subatomic particles to complex organisms - has some measure of *purusa* associated with it. *Purusa* establishes the combined entity's nature, its characteristics, and its behavioral possibilities. For example, an electron knows how to orbit (I am using the Bohr metaphor for atomic structure with full awareness that it is inaccurate. But it is illustrative in its own right) a nucleus because the entangled bit of *purusa* both knows how to do so and wills to do so. Although rare, it is possible for an electron to "act incorrectly" and thereby incur karmic consequences.

In more complex entities, especially biological organisms and very especially in human beings, the quantitative accumulation of purusa yields far more interesting behavioral possibilities. (Also greater potential for attached action and accumulation of karma.)

This kind of philosophy is not unique to Hindu and Buddhist cultures. Resonant ideas can be found in many cultures and philosophical traditions. Brigham Young, the colonizer and first Governor of Utah, advanced a very similar philosophy of matter infused with "intelligence." (As far as I am aware, without any contact with Hinduism.)

Christopher Alexander (of software engineering and, later, patterns fame) espouses parallel ideas in his newest works on the Nature of Order. His vocabulary uses "Life" instead of intelligence or purusa but, for him, Life infuses everything to one degree or another, the lesser the degree the lesser the "Quality Without A Name" and the greater the 'ugliness' of the construct.

The fact that purusa infuses all matter, at all levels - quantum to sentient, is justification for using a single invocation method (simple signals) regardless of the apparent complexity of the magical entity that is the target of the invocation. In fact, the apparent complexity of an entity (a human being, perhaps) is exactly that - apparent. Metaphysically speaking, nothing exists except Purusa and Prakrti so all invocations to apparent "purusa-prakrti" constructs are illusory.

Stimuli - intoning the Aum sound for example - utilize the imitative principle of magic to generate a single, simple, stimulus directed to the unary Purusa of an apparently complex entity (a human being). If the stimulus is correct, all the 'purusa' in the target attunes itself (vibrates sympathetically) with the stimulus resulting in a response of self-recognizing-Self.

Tantric sexual ritual (a kind of contagious magic) uses juxtaposition, physical contact, as the stimulus mechanism. Because the male and female entities involved in the ritual appear to be complex constructs a lot of simultaneous juxtapositions are required - but the ritual is nothing more than an aggregate of simple stimuli.

The relevance of this philosophy to our goal of seeking an alternative approach to computing is threefold:

- It adds a dimension of respectability and elaborates extensively on the simpler magical concepts of animism. The extensive exploration of the basic idea of *purusa* infused *prakrti* and how to interact with *purusa* is a fertile field for secondary metaphors within the umbrella of the Magic metaphor.
- It sets a constraint on how we conceive of "computing." Specifically, "computations" can be nothing more than the juxtaposed responses of an amalgamation of stimulated "purusa-prakrti-entities."
- Similarly, apparently complex stimuli (polyphonic instead of monotonic chants, yoga postures, Tantric sexual congress) must be simple aggregations of signal-stimuli.

Some development of the third point is in order. Purusa is akin to a "magical force" in that it has no "inside" - no intrinsic nature or structure. (Some Vedic philosophers might argue this point, but such esoterica is not relevant to our purposes here.) Purusa has nothing in common with our typical conception of a computer program. We cannot, therefore, think of magic in terms of a command line invocation of a stored and compiled program where that program can be arbitrarily complex in its function. We must think instead along the lines of stimulus and response mechanisms and conceive of programming only in terms of assembling an appropriate set of signals addressed to an appropriate set of forces resulting in an appropriate set of responses that, collectively, have the apparent structure of the ultimately desired result.

A variety of alternative illustrations of this stimulus-response concept are available to us. Consider but one, sympathetic vibration - a sound inducing a response in a properly constructed medium. Within the context of magic, an example might be the intonation of the Aum in an attempt to create a tonal stimulus that will resonate with the basic "frequency of the Universe" and therefore invoke a kind of harmony with that Universe. There is nothing programmatic about a tone and any resulting sympathetic vibration.

The constraint might be seen as overly restrictive - mandating creation of only the most basic and simple atomic responders or components. This is not the case. Complicated and multi-part stimuli are possible and are needful for many types of invocation. But the construction of these stimuli is not based on anything analogous to modular program design. Instead, they are analogous to music - the creation of a chord, or the polyphonic chants of Tibetan monks. Others are based on juxtaposition, like a sequence of notes, harmonies, or a well-turned poetic phrase. In all cases, the invocation has an evocative nature only - there is no element of representation, of computation or calculation, or of declaration (as understood in Lisp or Prolog programming). In all cases the response is reflexive with almost no element of reflection. (Some element of reflection does exist - hence the possibility of Karma, which requires willful action - but this subtlety is not essential to the main discussion at hand.)

Our discussion of magic and Vedic philosophy yields five points that will be revisited in our discussion of software development as "magic." In summary form, they are:

1. Users of magical things invoke/evoke responses that are intrinsic to the magical object.

2. All interaction between users and magical objects is of the form, stimulus-response.

3. Magical objects are limited to a specific set of responses.

4. Basic stimulus-response is completely dependent on the intrinsic nature of object being stimulated.

5. Arbitrarily complex responses can be evoked with comparably complex stimuli and that on both sides of the invocation, all apparent complexity results from juxtaposition (spatial or temporal) and not hierarchical decomposition.

At this point we have a metaphor and a philosophical position that can be used to support that metaphor. Before we can discuss application we need one additional element - a possible physical substrate to which we can apply the metaphor. For this last foundational piece, we will briefly examine current research in nano-technology - the concept of "Smart Matter."

Smart matter: bridging the mundane and the magical

"*Smart Matter is one of Xerox PARC's three cross-laboratory research themes. Smart Matter aims to exploit trends of miniaturization and integration of both computer hardware and micro-mechanical systems to build new kinds of machines. The idea is to trade computation (which is getting cheaper very fast) for physical or mechanical complexity. Some of its tenets are:*

- *Trading off computational and physical resources.*
- *Integrating sensing, actuation, and computation at fine granularity.*
- *Co-locating mechanical, computational, and electronic functions.*
- *Building systems with complex behavior from many simple pieces.*

As a research area, Smart Matter explores the "white spaces" among a wide range of disciplines: distributed computing, active control, robotics, software engineering, wireless communication, low-power electronics, smart materials, and MEMS."

John Gilbert, Principal Scientist
Xerox PARC

Xerox PARC (now PARC, Inc.) started the smart matter project at a fairly gross level of matter - a perforated board large enough to support a sheet of paper in need of alignment. At each hole in the board a jet of air could be used to create a force to align a sheet of paper. Also at each jet: a sensor to detect if the paper was above the jet. The sensors and jets were coupled to a computing device that digested the sensor input and output instructions to the jets to exhale, or not.

Subsequent and future efforts focused (will focus) on moving the computation closer to the sensor-jet dyads, perhaps with an analog of a neural net like connectivity so that the computation will be distributed across, and be a function of, multiple dyads.

A question asked by scientists engaged in this project, "how low (small) can you go?" At least one researcher anticipates nanometer scale smart matter. The nanytes of science fiction might very well be a commercial product, indispensable to your children and grandchildren.

Most of the research in the area of smart matter seems to make a basic assumption about the nature of computing in this type of environment. That assumption: essentially a replication at smaller and smaller scales of the typical computing environment in the macro-world. Specifically, creating small (perhaps special purpose) embedded computers communicating with each other via wired and wireless networks. Software for these environments would likely be familiar to any programmer of desktop and palm type applications and certainly to any embedded systems programmer.

In *Diamond Age: A young woman's primer*, Neal Stephenson writes of a world where the economy is based on nano-technology. Taking a cue from researches associated with Drexler, Stephenson assumes that nanytes will possess on-board mechanical (nanometer scale rods and springs) computers. A lot could be accomplished with this kind nano-scale device, but even more might be possible if we rethought how computing might (should) be accomplished in a radically new environment like a nanyte.

There is also an inherent limit to the scale at which you can still replicate anything like a computer and communication network. You certainly could not have computation occur at the level of a single atom or elementary particle following the prevailing notions of computing. In all fairness, it probably is not necessary to seek even nano-level computing. (And many people at PARC doubt it is possible.) However, a different approach to thinking about computing might make even gross scale smart matter simpler (and therefore much cheaper) at the same time it enables continued reduction in the scale of smart matter devices.

Smart matter research is exciting, not because it offers new insights into the possible nature of computing, but because it can create an environment - a kind of ultimate ubiquitous computing - that might be exploited by a new approach to computing arising from another area - or metaphor, like magic.

Smart matter provides a potential medium for applying the magic metaphor - one that is consistent with the Vedic metaphysics used to extend that metaphor. We are not ready to introduce some presuppositions or "first principles" that will provide a framework, or 'theory,' upon which a discipline of magical computing can be based.

A Theory of Magic

One principle, four premises, and three corollaries comprise, at present, a theoretical framework or foundation for magical computing. Additions to this foundation are likely (if anyone is captivated enough by the metaphor to explore it in more depth) but the elements presented here must be considered as a mandatory set. Consistency and conformity to these elements - in their entirety - is prerequisite to making any claim to be a "magical software technology."

Principle One: Ward Cunningham's, "The simplest thing that could possibly work." William of Occam proposed a very similar principle as a tool for deciding among competing theories. Ward's formulation is better suited to dynamic decision-making. Whenever we confront alternatives - theoretical or applied - we will opt for the simplest choice possible. This will become particularly relevant when we work on "casting spells" (programming) and we are confronted with temptations to introduce complications in order to achieve "flexibility" or "compatibility." Adherence to Principle One mandates resistance to such temptations.

Premise one: Intelligence (purusa / spirit / life / computing) can be distributed across the entire spectrum of potential platforms, from atoms to von Neumann architecture computers to human beings if, and only if, two conditions are met:

One, the same mechanisms and principles apply at all levels, micro to macro, and that mechanism is simple stimulus – response. Stimuli and responses are simple signals, no information content (remember principle one). This does not mean that a stimulus or a response cannot have complicated form. It merely means that, however complicated, stimuli and responses are never anything other than aggregations of stimuli juxtaposed in space or time.

For example: It is easy to think of a single tone as a signal with no content. It is tempting, however to see an orchestral performance as being somehow qualitatively different. Obviously it is not. A performance is nothing more than a collection of single tones juxtaposed in time (simultaneous or sequential) and, to a lesser degree, space - origin points are arranged in a prescribed manner. The purpose of the amalgam of notes is the evocation of a response in the listener(s).

Two, responses are totally dependent on local resources – the entity receiving the stimulus can respond only on the basis of its own state, its own intrinsic nature. You cannot coerce a magical entity to respond differently than its nature allows by passing arguments (signals only, remember). A magical entity cannot supercede its own nature by collaborating with other entities. No magical entity is dependent (especially in the sense of dependency familiar to modular software developers) upon any other magical entity.

The purpose of these restrictions is easier to see when one thinks of computing at the level of a single atom, but seem unnecessarily restrictive at macro levels. For the moment, the only rebuttal arguments are: Ward's first principle; and, "we are looking to redefine computing, not merely rehash some aspect of that discipline."

Corollary one: Representation – one of the two Cartesian (Rationalist) foundations for computation as we currently understand it – is not our friend! Neither stimuli nor responses 'represent' anything – they just are. Again, this is easier to see at a micro-level but much harder at, say, a human level where we like to believe that we are symbol processors in addition to being subject to stimulus-response behavioral patterns. Whatever the case of humans might actually be (an there are reasons to believe that stimulus-response plays a far larger role in cognition that most are willing to believe) – magical computing will be restricted to the evocative, not denotative, realm.

Corollary two: stimulus-response is as simple as computing based on binary logic without being as simplistic. Stimuli and responses can be arbitrarily complicated but remain non-parsable, hence without losing their status as signals. (Parsing, in this context, means you cannot decompose a complicated signal into components with differing semantic meaning. A complicated signal can be separated into discrete simple signals but that separation provides no additional meaning, since it is the combined signal that is the stimulus that is required to invoke a response.)

This gives us a much more varied and interesting set of 'building blocks' from which to construct computation without incurring any of the costs associated with Turing machines. (One example of such costs: in theory, it is possible to construct a representation of the universe as a string of 1s and 0s. Also theoretically, a program – itself a sting of 1s and 0s, could be constructed and applied to the first string to simulate the dynamics of the Universe. But, construction of either string and execution of the program – all would take longer time than the Universe itself has existed. Something more direct is required if we are to achieve magical computing.)

Corollary three: although much of the language of stimulus-response implies some kind of media-based exchange [like a flow (stimulus) of electronic voltage (medium) evoking a tone (response)] there are other categories of stimulus and response. Geometry, for example, can be a stimulus and a response – like the docking of molecules or biological organisms based strictly on the geometry of their structure. Stimuli and responses need not be in the same category in order to participate in any given stimulus-response construct. Consider the guitar chord where a given response is determined by three stimuli: length of string (geometric), tension (static-force), and stroke (dynamic-force) synthesized despite being of different categories.

Premise two: Intelligence (purusa / spirit / life / computing) implies "willful being-ness." Another way of stating this, "everything has a motive to exist and to participate in existence." This premise is primarily metaphoric. Whether or not it is literally true is irrelevant for our purposes. When we discuss design of magical entities and incantations (spells) we will want to use an anthropomorphic principle (as was the case in behavior driven objects) as a constraint on our thinking. Premise two is therefore a basis for mental discipline.

Premise three: complicated stimuli and complicated responses come about from the application of two mechanisms: synthesis and juxtaposition.

Synthesis is the seamless integration of multiple stimuli (or responses) into one. Perhaps the best example is a chord that produces a single tone (response) via the simultaneous application of three stimuli – tension, length, and stroke – to a single entity (a guitar string perhaps).

Synthesis is important because it allows us to create intermediaries - magical entities that respond to stimuli by producing an output that we can use as a stimulus to some other magical entity.

Juxtaposition is nothing more than the congruence of stimuli and responses in terms of space and/or time. A chord is an example of synthesis because the stimuli are integrated to evoke a specific single response. The sound of an orchestra at a discrete interval of time is the result of juxtaposition of discrete notes produced during that interval. A musical phrase is an example of juxtaposition in that its overall evocative power (its ability to function as a stimulus) results from the sequencing of discrete stimuli over a time interval.

A special case of juxtaposition would be the combination of two or more magical entities in order to modify the responses of one or both the conjoined entities. Juxtaposing a volume of water and a container (a jug perhaps) changes the "value" of the response evoked by the passage of air over the mouth of the container. (Of course, it is important to juxtapose the inner surface of the container with the volume of water rather than the outer surface if the desired result is to be achieved.)

Synthesis and juxtaposition provide a means to achieve complicated structure without the concomitant implication that such structures can be pre-determined or "engineered." The success of a musical phrase, the ability of a chord to evoke a response is not determinable except via experimentation and after-the-fact analysis.

Premise four: It is possible for multiple entities to instantiate systems of cooperation and coordination (via juxtaposition and synthesis), but control is both infeasible and undesirable. It is also possible for an environment to provide coordination and enhance cooperation by existing as a patterned or persistent 'stimuli zone."

An example of absence-of-control cooperation would be the "structural coupling" described by Maturana and Varela in their "new biology" based on autopoiesis.

An example of environment-based coordination would be a magnetic field that provides a consistent and persistent stimulus to which iron atoms respond by changing their spatial orientation. "Field" type environments could be as simple as magnetic fields or as *outré* as

Rupert Sheldrake's morphogenetic fields, or as complicated as David Bohm's and Karl Pribram's quantum and holographic fields, respectively.

Practical Magic

At this point we have a metaphor, "magic," a kind of theory or explanation of how to think about magic, and a substrate or physical platform, nanotech, which can be "enchanted." We can also summarize the main points discussed so far:

- Magic is an evocative process – a kind of stimulus-response mechanism. A "spell" is the evocative stimulus.
- The ability of an "enchanted" object to respond to stimulus is not algorithmic or programmatic in nature.
- Enchanted objects can have complicated structure, as can spells, but that structure reflects nothing more than the juxtaposition of responses, or stimuli, in space and/or time. Synthesis – juxtaposition that results a qualitatively different thing (like hydrogen and oxygen juxtaposed in a specific way to create water) – is a possible consequence of juxtaposition.
- Enchanted objects are totally and absolutely autonomous. Even when aggregated or synthesized, there is no organization and there is emphatically no control of one object by another.
- Enchanted objects can cooperate with each other by a process of autopoietic organization based on the exchange of stimuli and responses. (Maturana's and Varela's *Tree of Life: a New Science of Biology* provides insights into how this simple mechanism can generate complex cooperative communities of objects.)
- 'Fields,' analogous to magnetic fields can result from the aggregate responses of a collection of enchanted objects (the atoms in a copper winding) to a common stimulus (application of an electrical current) and such fields can provide a common stimulus to a collection of enchanted objects.

The value of any new metaphor derives from its utility. Utility can arise from some kind of implementation – a new language, library, or artifact, for example – or, as in this case, by suggesting some concrete topics for further exploration. If such explorations prove to be fruitful in any kind of pragmatic manner then the metaphor is a good one. Some research topics:

1. Is there a finite and enumerable set of "primitive" enchanted objects from which everything else comes into existence via juxtaposition and synthesis? Remember that everything in the Universe results from the juxtaposition and synthesis of a finite and pretty small number of elements. (Or a still smaller number of fundamental particles, or a smaller yet number of quanta.)
2. Is there a way for these primitive enchanted objects to interact with each other with resulting complicated (perhaps complex) macro-objects whose enchantment is qualitatively different from the enchantments of any individual primitive? Autopoiesis and biology suggest the answer is yes. Using the magic metaphor as a lens for exploring biological metaphors of computing will likely yield quite different results than current efforts to frame computing in biological terms or biology in computational terms.
3. Can we devise a "science" of enchanted object juxtaposition and synthesis? The goals of such a science would be to create new and useful enchanted objects capable of providing a response desirable for human beings. Such a science would be much more analogous to

chemistry and the culinary arts than it our current understanding of computational science.

4. Can we discover patterns in the autopoietic organization of enchanted objects that would offer insights and shortcuts to support our new science of juxtaposition and synthesis. It seems likely given the work of researchers as diverse as D'Arcy Thompson and Christopher Alexander.
5. To what extent can geometry provide a formalism in support of our science of juxtaposition the way that algebra, logic, and various calculi have provided for contemporary computer science.
6. Can we think of user interfaces in terms of "amulets" – magical objects that exist primarily to translate stimuli created by humans into stimuli that can evoke behavior in magical objects? An analog for this kind of translation – the way that rubbing the rim of a crystal bowl (an stimulus that a human can provide) evokes a tonal response that is beyond the capability of a human voice to produce directly.
7. How would be go about enchanting ordinary objects (doors for example) so that they would respond to simple incantations like, "open sesame?" (Perhaps, by juxtaposing a thin layer of magical objects that reverse some kind of polarity in response to the sound vibrations of our voice.)
8. Can we think about "demons" (somewhat similar to those found haunting operating systems) as a special kind of mediator between humans and the magical world? A demon would be capable of responding to a simple, human generated stimulus, by finding other magical objects and uttering appropriate incantations to them on our behalf. Demons would be a magical way to encapsulate our current understanding of algorithmic computing in the sense that a program is just the juxtaposition of a set of discrete imperatives (spells).
9. Can we devise an enchanted world where everyone can create the auditory or kinesthetic stimuli that evoke appropriate everyday responses in support of human activities? Wizards would be specialists that had memorized more complex spells necessary to evoke sophisticated and special purpose responses from that world. Shamans would be the most advanced magical practitioners – capable of creating as well as manipulating magical objects.

Conclusion

"Papers in the Onward! Track are not aimed at advancing the state of the art - they're aimed, instead, at altering or redefining the art by proposing a leap forward - or sideways - for computing."

Hopefully this paper provides some ideas curious enough and sufficiently 'sideways' that they will magically evoke more and better ideas completely outside the framework provided by contemporary computer science and software development paradigms.

Notes on Postmodern Programming

James Noble, Robert Biddle
Computer Science,
Victoria University of Wellington, New Zealand.
{robert,kjx}@mcs.vuw.ac.nz

October 29, 2002

0 Manifesto

The ultimate goal of all computer science is the program. The performance of programs was once the noblest function of computer science, and computer science was indispensable to great programs. Today, programming and computer science exist in complacent isolation, and can only be rescued by the conscious co-operation and collaboration of all programmers.

The universities were unable to produce this unity; and how indeed, should they have done so, since creativity cannot be taught? Designers, programmers and engineers must once again come to know and comprehend the composite character of a program, both as an entity and in terms of its various parts. Then their work will be filled with that true software spirit which, as "theory of computing", it has lost. Universities must return to programming. The worlds of the formal methods and algorithm analysis, consisting only of logic and mathematics, must become once again a world in which things are built. If the young person who rejoices in creative activity now begins his career as in the older days by learning to program, then the unproductive "scientist" will no longer be condemned to inadequate science, for their skills will be preserved for the programming in which they can achieve great things.

Designers, programmers, engineers, we must all return to programming! There is no essential difference between the computer scientist and the programmer. The computer scientist is an exalted programmer. By the grace of Heaven and in rare moments of inspiration which transcend the will, computer science may unconsciously blossom from the labour of the hand, but a base in programming is essential to every computer scientist. It is there that the original source of creativity lies.

Let us therefore create a new guild of programmers without the class-distinctions that raise an arrogant barrier between programmers and computer scientists! Let us desire, conceive, and create the new program of the future together. It will combine design, user-interfaces, and programming in a single form, and will one day rise towards the heavens from the hands of a million workers as the crystalline symbol of a new and coming faith.

1 To Our Reader

These notes have the status of “Letters written to ourselves”: we wrote them down because, without doing so, we found ourselves making up new arguments over and over again. When reading what we had written, we were always too satisfied.

For one thing, we felt they suffered from a marked silence as to what postmoderism *actually is* [77, 9]. Yet, we will not try to define postmodernism, first because a complete description of postmodernism in general would be too large for the paper [62, 44, 64, 73], but secondly (and more importantly) because an understanding of postmodern programming is precisely what we are working towards.

Very few programmers tend to see their (sometimes rather general) difficulties as the core of the subject and as a result there is a widely held consensus as to what programming is really about. If these notes prove to be a source of recognition or to give you the appreciation that we have simply written down what you already know about the programmer’s trade, some of our goals will have been reached.

2 On Our Ability To Do Much

We are faced with a basic problem of presentation. What we are really concerned about is the composition of large systems, the text of which may occupy, say, a significant fraction all the digital storage media in the known world [74].

Our basic problem is simply the success of modern computer science. History has shown that this truth is very hard to believe. Apparently we are trained to expect a "software crisis", and to ascribe to software failures all the ills of society: the collapse of the dot-com bubble [28, 31], the bankruptcy of Enron [51], and the millennial end of the world [78].

This corrosive scepticism about the achievements of programming is unfounded. Few doom-laden prophesies have come to pass: the world did not end with fireworks over the Sydney harbour bridge, and few modern disasters are due to software. To consider just two examples: the space shuttle crash was not caused by software — indeed, Feynman praises the shuttle software practices as exemplary engineering [24]; and the Dot-Com Boom (like the South Sea Bubble) was not caused by failure of technology, but the over-enthusiasm of global stock markets.

In short, one cannot be woken up in the morning, travel to work, listen to radio or music; watch television; play games; speak or TXT down the 'phone; read newspapers or books; write conference papers, journal articles, government or corporate reports; save or spend money; buy food, cook it, order or pay for it at restaurants ranging from McDonalds to the Ritz; without every activity critically depending upon the results of programming. These programs are not perfect: but neither are they the complete, expensive failures beloved of armchair critics, whose behaviour belies their own rhetoric whenever they fly across the Atlantic in automated aircraft to speak at conferences and then use Internet banking to check their accounts. The measure of software is our irritation at its failures, not our surprise that it works at all.

Summary: as quick-witted human beings we have built very large computer systems and we had better learn to live with them and respect their limitations while giving them due credit, rather than to try to ignore them, so that we will be rewarded by continued success.

3 On the Notion of Program

The object of study of computer science is the program — somehow cobbled together by old-time programmers; measured, metricated, analysed, theorised, critiqued, tested, compiled, optimised, and often ignored by modern computer science. We consider that the term "program" is both too big and too little for postmodern computer science.

"Program" is too big because often we are working on parts of programs: objects, functions, classes, components. We may have no idea which (if any) program these parts will end up part of.

"Program" is too small because often we are working on multiple programs: perhaps the small component is to be part of many large programs; perhaps there is a framework or library that will be reused many times; perhaps our program must communicate with other programs, who in their turn communicate with yet more programs, so the subject of our endeavour is somewhere this interconnected network.

Large or small, the quality to which we refer is perfectly precise. Like bad art, we know it when we see it. Still, it cannot be named.

A word which we most often use to talk about programs is "component".

Yet, a component can only be a subpart, not a whole.

Another word which we use to talk about programs is "system".

A system could be large, or small, but includes the strong connotation of systematic, systems theory, system thinking, that the system is organised, rationally subdivided, structured recursively into a tree of modules, modern. The word "system" is too enclosed.

The word "algorithm" is often claimed as the central concept of computer science [35]

"Algorithm", however, leaves out large amounts of the discipline of programming: components, patterns, protocols, languages, data structures [76].

The word "software" is reminiscent of undergarments.

The phrase "*the software without a name*" could capture precisely what we wish to address. Unfortunately, experience teaches us that this software would soon have a name: "*SWAN*".

Thus, we have retained the word program, but treat it as under erasure, as meaning whatever program, program subcomponent, or supersystem we happen to be working on at the time.

4 On Pervasive Heterogeneity

Modern computer science dreamed of the personal computer: one machine usable by one person running one application written in one language. The personal computer was "the computer you could unplug".

We have progressed far beyond the modern dream. There is a "Computing Rainbow" [33] — an interdependent system of global computation with a multitude of machines supporting many languages, applications, and users, with heterogeneous architectures and differing capacities, costs, prices, and ownership. You cannot unplug computers even if you could want to: why would you want to avoid reading email from your friends; disconnect the full authority digital engine controllers from the turbofans carrying the airliner; or remove the cochlea implants correcting your congenital deafness?

Each of these subareas of programming has their own concerns, forces, difficulties, problems. Computer science is itself fragmented, although some concerns cut across several areas. But this heterogeneity does not operate only at the abstract level of the field as a whole; rather programs themselves are increasingly heterogeneous. A program may include a PalmPilot client which interfaces directly to an 360-architecture mainframe; Windows XP microcomputers may feed information into a Sun minicomputer; one division of a company may run Compaq VMS systems while another runs IBM AS/400 or HP7000. Programs have to federate across diverse systems, without any common language, protocol, or necessarily even character set in common.

5 On Abstraction

At this stage we find it easy to say something about the role of abstraction, partly because it permeates the whole subject.

Modern computer science describes the relationships within programs as "abstractions" — we may say an object in a program is an abstraction of the real world [8].

Computer scientists and mathematicians are familiar with abstractions: for example, a stack is an abstraction that might be implemented by an array, a pointer, and some executable code; the stack is an abstraction because it elides many of the details of actual implementation [17].

Unfortunately, it doesn't seem to make much sense to say that a Bovine object in a program is an "abstraction" of a real cow in a farm in this way: it doesn't make sense to say that the object in the program is "implemented" by a cow in reality, or that the objects in the program are special kinds of cows which do not eat, excrete, or expire. Alternatively, following Plato, we could have an abstraction of a cow as the "ideal, immutable, eternal form" of a cow, perhaps corresponding to a Cow *class*, but, again, this kind of abstraction is not a good description of the relationship between the cow object and the real cow [68].

Postmodern computer science proposes a range of different descriptions of the relationship between programmatic object and external object. For example, that this kind of relationship can be seen as semiotic: that is, the object in the program can be seen as a *sign* of the object in the world [55, 4, 3, 2]. Unlike abstractions, which can be reasoned about using deduction (from causes to effects), signs are effectively implications, and are modelled using abduction (reasoning from effects to causes) [21].

Semiosis may also support developments of a theory of debugging (determining bugs from symptoms); of analysis (determining the programs from requirements); and a metatheory of design (determining patterns, algorithms, structures from concrete programs) [54].

6 On Requirements

Postmodern computer science holds that no requirements can be both complete and consistent: you have to pick one.

Descriptivists, postmodernists choose completeness over consistency. In analysing a system (for example with Usage-Centered Design [13]) we may consider multiple users providing multiple requirements, and distinguish between actors who will finally use the system, customers who pay for it, clients who commission it, and stakeholders who wish they were involved — all of whom may be in conflict. Postmodern analysis uses techniques to handle inconsistency, such as iteration on designs, prioritising, and lying where necessary [61].

In contrast, modernists choose consistency over completeness. To perform any traditional formal analysis (without resorting immediately to modal logics) a description has to be consistent: before forming such a description, much information must be elided to ensure this consistency — adopting such a frame of reference necessarily excludes information that lies outside. Of course, a consistent definition has the great advantage of automated or manual checking, often the aim of the modernists — but as postmodernists we are willing to coopt their techniques whenever we feel they are useful.

Computer scientists with a formal bent often claim that design and implementation are an exercise — in the refinement calculus you can gradually transform a problem statement into a program, maintaining correctness at every step: problem frames have similar potential. From a postmodern perspective, however, such a definition is not a problem statement, but rather an abstract definition of a solution — where refinement simply makes a solution more concrete. Similarly, Jackson's problem frames are selected to match particular solutions, rather than problems per se: one fits problems to frames, rather than adjusting frames to enclose problems [43].

Corollary of this section: Formal analysis can be used to show the absence of bugs, but never to show the correctness of the specification. Or, to quote Alan Perlis: *"102. One can't proceed from the informal to the formal by formal means"* [59].

7 No Big Picture

A key characteristic of postmodernism is the absence of a "grand narrative" [64, 50, 62]. Where once the majority of the world would have believed in God or Marx, where architecture was simply building steel framed rectangular glass boxes, where music was constrained to the twelve-tone row, where citizens of a country all spoke the same language and supported the same cricket teams, we may now contemplate with fear a sea of chaos, perhaps held together by unenlightened self-interest.

Consider the Internet as we know it: a connection of loosely coupled computer systems, of varying capacities, architectures, ownerships, costs, and sizes. It connects every conceivable variation of every operating system and every computer. It speaks many network protocols (HTTP of various versions and brands; Telnet; POP; IMAP; NETBUI; AFS; SMB; ...); and through hardware software gateways, the systems and protocols of what appears to be the Internet are in fact unlimited. It can even reach computers that have been obsolete and no longer exist: retrocomputing lives through emulation and lives on the Internet.

Even the user experience of the World Wide Web is extremely diverse: every web site has its own design, its own interaction style, its own personality, with no commonality other than the menu bar provided by an individual's browser, one of many available, and customisable on a whim.

Ward Cunningham's Wiki Wiki Web [15, 49] shows that even the historic success of the World Wide Web left room for improvement: the Wiki design involves exquisitite light touch, and allows the Wiki to be many things at once, all of them dynamic, collaborative, and — healthy.

Compare this all with representations of a computational "infosphere" in popular science fiction — such as *The Matrix* [71] or *Neuromancer* [34]. These are typically modern in character, working in a complex but coherent way, and presenting a uniform interface: the "Matrix" presents a realistic single graphical presentation common to all users. Ironically, the postmodern Internet is more real than these fantasies; and there is no one viewpoint on the Internet, and there may be no commonality between two web sites even if hosted on the same server and designed by the same people.

Although it clearly developed from the original success of the modern design of the Arpanet, the success of the Internet now is postmodern in character. But it is success. And the tolerance of eclectic diversity is a key cause of the success of the internet: it has allowed growth and interaction instead of isolation and alientation.

For postmodern programming, the absence of an overarching grand narrative means eclectic tolerance in programming terms:

- There is equal acceptance of high and low culture: Visual Basic and Haskell are equally of interest, as there is no reason to applaud the one and disparage the other.

- The past is just another part of the present — programs can call on elements of modernism, either aesthetics or technology, and combine them together in equal measure. As ancient computers live through emulation on the Internet, so ancient programs and languages can live in connection with programs not yet written.

- Programs can exhibit "faults in construction" that would be forbidden by a modernist approach.
- Programming techniques (such as design patterns) and systems (such as Aspect/J [45], or even the continuation code sections in literate programming [46]) explitly support program organisation involving communicating diverse elements.

Without a grand narrative, there will not be one common way to program, or even one common kind of interface between programs. The alternative is the postmodern multidimensional organisation encompassing many little narratives.

In practice, narratives may grow and shrink, reflecting the exercise of power (especially by monopolist organisations) and the development of communities (especially where cooperation is mutually beneficial).

Moreover, there are many kinds of narrative in programming, and systems may have a postmodern character in some aspect but modern character in others. For example the Microsoft Common Language Runtime [36] is postmodern in that it supports a large number of programming languages — modern (C), postmodern (Perl), and historical (COBOL), high culture (Haskell) and low (Visual Basic), with access to low-level features as necessary (a modernist would consider this a fault). However, it is modern in that it achieves this by enforcing a common bytecode format — indeed, a particular subset of the format, and that it deploys the apparatus of power (verifiers, compliance kits, bytecode type checkers, developer certification, code component certificates) to enforce the commonality.

8 On Modular Components

Modularity, and interchangeable modular components are a key component of the modernist approach in software, as in architecture, marketing, production, and elsewhere.

Postmodernism admits modernism as one mode of expression, so modular or generated components (as with other modern techniques or tools) can readily be used to postmodern ends.

Consider for example the Sydney Opera House. This building is now symbolic of Sydney and Australia, but was originally sketched on the back of a napkin by Jørn Utzon. The key feature of the Opera house are the shell-like roofs above the Opera Theatre and Concert Hall (a design which is postmodern because it has nothing to do with the function of the concert halls below).

The construction of the Opera House is only possible because the distinctive shells come from a single geometric spherical section. This was not part of the napkin design — yet according to Utzon, this “solves all the problems of practical construction by opening up mass production” both for the tiles on the roof, and the ribs supporting them. Without mass production of modular components the Opera House could not have been completed.

9 No Metaphor

Postmodern programming rejects overarching grand narratives.

As a result, it favours descriptive reasoning rather than prescriptive. Rather than working top down from a theory towards practice, postmodern programming theories are built up, following practice. Moreover, theory follows practice on a case-by-case basis — "the world is all that is the case" [77].

Note that here we don't necessarily mean "theory" in the mathematical sense of theoretical computer science. Rather, we mean theory in the traditional speculative sense that serves to help us organise our experiences. For example, theories of how best to program would include stepwise refinement, object-orientation, and pattern languages.

Postmodern programming limits the scope of theory (and formalism) to particular "little narratives" — conditions where that theory is applicable, or is generated by the practice. This limits the kind of questions that will be asked of theory, and theory's position within the discipline as a whole.

Many metaphors have been adopted to describe programming: computer *science* (hypotheses, experiments, research); software *architecture* (plans, building, implementing); software *engineering* (design, verify, construct); and we may see programs as *literature* (write programs, literate programming); programs as *evolutionary biology* (program evolution, cellular automata); programs as *neurobiology* (artificial neural networks); programs as *mathematics* (programs as theorems, as proofs, as type systems).

Within modern computer science (following modern architecture, or disciplinary inadequacy due to the low status of computer science departments in many universities) there is an intellectual posture that accepts metaphors from other disciplines uncritically [65], without providing arguments as to why that metaphor should be applicable [79]. In general, this is the result of the modern grand narrative: computer science must conform to some theory, where that theory is carried by metaphor: the program as proof, as bridge, as house, as city.

Postmodern computer science tends to eschew metaphor — rather, in place of a metaphor we have a past. This is true on both the large scale (the discipline as a whole) and the small scale (individual programmers and programs). Postmodernism is often descriptive: recording the state of the world, rather than presenting some grand theory. Writing programs follows reading programs, because postmodern programming is extension, recovery, reuse, rather than creating masterpieces from nothing. Theory follows practice, because we aim to understand the world as it is, rather than remake it from scratch with a genesis device [66].

Our view is that computer science has "come of age". Computer Science is sufficient for itself: albeit as an 'unrestricted science' from where investigators must be prepared to follow their problems into any other science whatsoever (Pantin, quoted by Becher & Trowler,[5, p.32]). That is, we think it sensible that related disciplines are applied to their domains, so physics is used to address the design of semiconductors, statistics to analyse web server performance, accounting to study e-commerce, semiotics and psychology to drive human interface design, or linguistics to categorise Visual Basic programs. Of course we should be prepared to learn from many other disciplines. But the program itself the ultimately the subject of computer science itself.

10 No Future

What is **post-**modernism? And where does it lead? How can something be after what is **modern**?. Isn't modern what we have *today?*

Modernism is a term used to describe a range of developments in architecture, literature, philosophy, and then society generally. Postmodernism is what comes after modernism. The question is, does postmodernism:

- **replace** modernism?, or
- **fulfil** modernism?

Inasmuch as there is an answer, it is both. (This is the standard postmodern answer to any question. 'Tsall good). Postmodernism is a replacement for modernism because the postmodern theories or practices replace the modern. Postmodern architecture has replaced modern architecture; postmodern fiction has replaced modern fiction; postmodern programming languages (Perl, late C++) replace modern programming languages (Pasal, ANSI C).

But postmodernism (or postmodernity, the society and culture that follows after modernity) is simultaneously the fulfilment of modernism. Without the technology developed by modernity, there could be no postmodernity or postmodernism. Thus, Extreme Programming, for example, aims to replace modern and late-modern methodologies (e.g. Responsibility Driven Design or the "Booch" methodology, and the Rational Unified Process or the OPEN process (now deceased)) [6, 75, 7, 48, 37]. On the other hand, XP also claims to be the fulfilment of a number of modern movements: including rigorous testing, consistent coding and naming style, and late-modern programming languages and environments (e.g. Smalltalk) perhaps with postmodern extensions (JUnit, the Refactoring Browser). Similarly, postmodern programming does not reject but rather embraces elements that are themselves the ultimate products of modern development.

11 Perl, The First Postmodern Programming Language

What is a postmodern programming language? Or, what is a modern programming language? The second question is easier to answer than the first: a modern programming language supports a single (modern) style of programming, based on recursive decomposition of both code and data.

Modern programming languages can themselves vary in a number of ways. Common Lisp, APL, and Smalltalk, for example, are based on difference approaches to programming, but all are modern, and all rely on extensive support for their programming theories. Pascal, Oberon and Scheme and Self are more minimalist counterparts, but also modern.

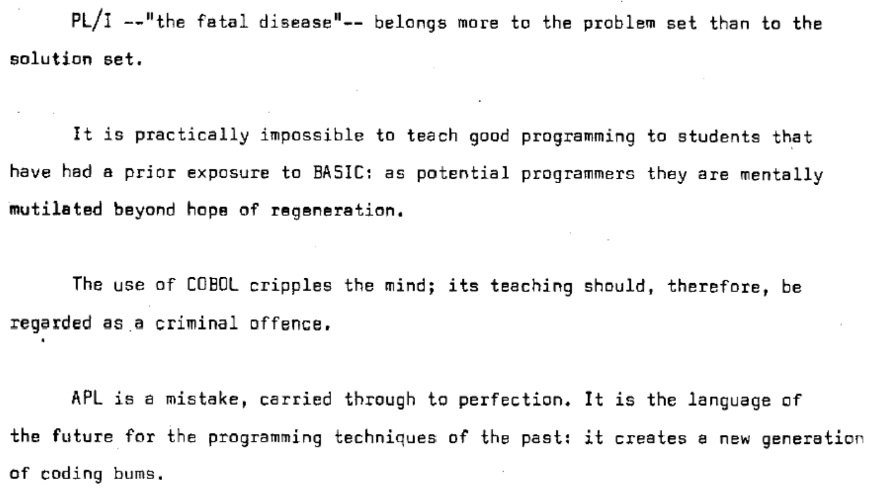
Larry Wall explains how his design of Perl was explicitly based on a postmodern approach. This explanation shows how postmodernism, easily misuderstood or disregarded by pragmatists, is responsible for a programming language highly prized and defended by those same pragmatists. Wall's reasoning deserves to be read in in its entirety, but the key point is that the design was not based on any grand narrative, but on a case-by-case basis: [72].

> *When I started writing Perl, I'd actually been steeped in enough postmodernism to know that that's what I wanted to do. Or rather, that I wanted to do something that would turn out to be postmodern, because you can't actually do something postmodern, you can only really do something cool that turns out to be postmodern. Hmm. Do I really believe that? I dunno. Maybe. Sometimes. You may actually find this difficult to believe, but I didn't actually set out to write a postmodern talk. I was just going to talk about how Perl is postmodern. But it just kind of happened. So you get to see all the ductwork.*
>
> *Anyway, back to Perl. When I started designing Perl, I explicitly set out to deconstruct all the computer languages I knew and recombine or reconstruct them in a different way, because there were many things I liked about other languages, and many things I disliked. I lovingly reused features from many languages. (I suppose a Modernist would say I stole the features, since Modernists are hung up about originality.) Whatever the verb you choose, I've done it over the course of the years from C, sh, csh, grep, sed, awk, Fortran, COBOL, PL/I, BASIC-PLUS, SNOBOL, Lisp, Ada, C++, and Python. To name a few. To the extent that Perl rules rather than sucks, it's because the various features of these languages ruled rather than sucked.*

Perl is a success, and there is no grand narrative for its design. However, over time Perl is getting more modern, accumulating all the trappings of a modern language: objects, packages, namespaces, syntax, etc. Again, this is an example of the post-modern trait of absorbing technology from modernism while putting it to a rather different use.

While Wall's rationale is explicit, other languages can place a good claim to be considered postmodern, at least in some aspects. Hypercard and Visual Basic extend programming to include forms of multimedia and graphical user interface (again without getting some religion: Prograph is modern in its insistence on the primacy of visual syntax). Intercal must be considered as a post-modern language (mostly for non-technical reasons).

PL/I could be considered post-modern, as it was designed to support Fortran, Algol, and even COBOL programming styles, although (unlike Perl) it attempted to bring them all within a modern unified framework. This mixed result is involved in Dijkstra calling PL/I a fatal disease [18]:



PL/I --"the fatal disease"-- belongs more to the problem set than to the solution set.

It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

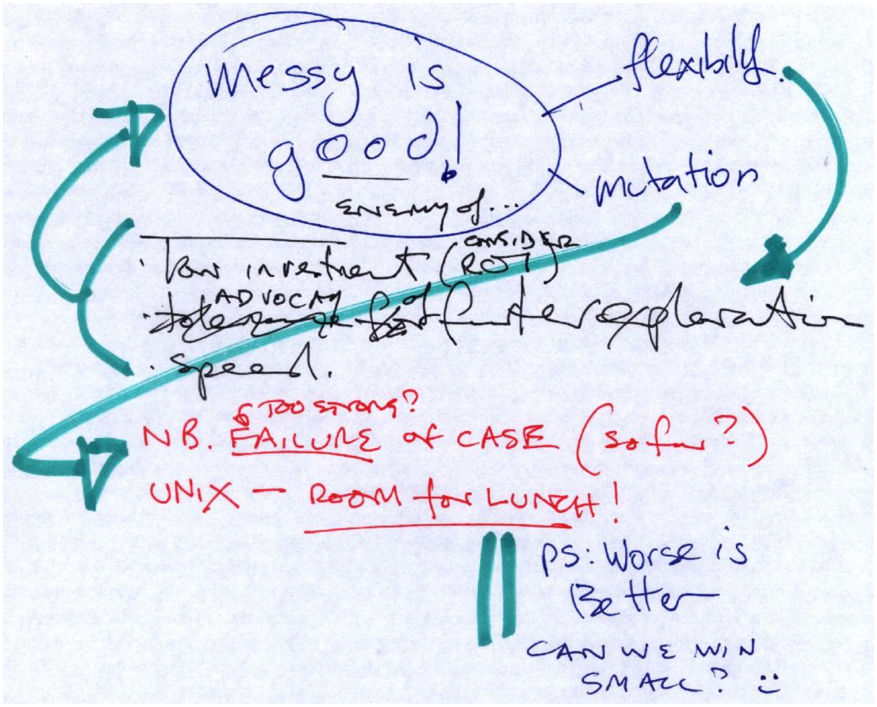
The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

C++ is an interesting case. Early C++ — C with Classes, the C++ used in the *Design Patterns* book is essentially a modern object-oriented programming language: classes are the primarily (and only) modelling technique, no-one seriously advocates the procedural features of a language — or rather, these "faults in construction" are tolerated as faults within the modern narrative. As C++ evolves, into what we call "Late C++" (templates, exceptions, dynamic casting) the discourses surrounding C++ change also — leaving the object-oriented grand narrative, and becoming postmodern, building a theory of multi-paradigm design and programming upwards from the language features [14, 67, 47]. While *Design Patterns* (as one data point of C++ circa 1994) has essentially no examples of C++ free functions, by the late 2000 Late C++ is clearly advocated as a postmodern language.

Finally, it is interesting to consider Java vs. C♯. Both are arguably postmodern languages, although less so than Perl, and with stronger streaks of modernism, especially in the one-language rhetoric surrounding Java, matched by the CLR rhetoric surrounding C♯. There are no significant technical differences between the two languages — both with C++ syntax, somewhat moderated by the Pascal tradition, with a ersatz-Smalltalk object model and a handful of Modula-3 thrown in for concurrency and modularity. The key reason these languages are postmodern is that they cannot be considered against technical criteria: comparing them is like comparing Pepsi and Coke: you don't drink the cola — you drink the advertising [69].

12 Messy Is Good



Messy is good!

ENEMY of...

flexibility.

mutation

· Low investment (CONSIDER ROI)

· ~~tolerance~~ ADVOCACY ~~for~~ of further exploration

· speed.

N.B. FAILURE & TOO STRONG? of CASE (so far?)

UNIX — ROOM for LUNCH!

PS: Worse is Better

CAN WE WIN SMALL? :-)

13 A First Example of Scrap-Heap System Construction

In the section "No Metaphor" we will have stressed that postmodernism has a past, and that this past is reflected in both the structure of the discipline and the practice of programming. In particular, this past exists as a large number of existing programs that the postmodern programmer can scavenge through and reuse.

Instead of presenting (as a ready-made product) what we would call a scrap-heap program, we are going to describe in detail the process of creating such a program. We do this because many programs are just there: they do not have to be made, and the kind of programs we are particularly interested in are those which we feel to be comfortably outside our powers of construction and conception.

The task is to instruct a computer to print a table of the first thousand prime numbers, 2 being considered the first prime number.

To write this program, we first connected our computer to the Internet, downloaded some music from Napster, and then read our email. (You have to receive email to perform a workday [11]). We received 25 pieces of email of which 16 were advertisements for Internet pornography, administrivia, or invitations to invest in Nigerian currency trades. After dealing with this email, we typed "calculate prime numbers" into Google. This found several web sites regarding prime numbers, and some more pornography. After a while, we were interrupted, and so moved on to the prime number web sites. In particular, `http://www.2357.a-tu.net` includes the "ALGOMATH" C library for calculating prime numbers; another site included an EXCEL macro which was to complex to understand. Although we had not programmed in C for years, after downloading and compiling the library (by typing `make`), we noticed the documentation included the following program:

```
int *pointer , c=0;

if((pointer = am_primes_array(4, 3)) == NULL)
    printf("not enough memory\n");

while( *(pointer+c)){
                   printf("%d\n",*(pointer+c));
       c++;
   }

return;
```

We cut and pasted this program into a file and compiled it several times, having to add a few extra lines (e.g. `main () {`). Eventually we ran it, and indeed it appeared to generate three prime numbers larger than four. We edited the parameters to `am_primes_array` to `(2,1000)`, and then ran the output through `wc -l` to check that it had printed 1000 numbers.

Here we have completed what we announced at the beginning of this section, viz. "to describe in very great detail the composition process of such a [postmodern] program".

14 We're all Devo

- The Open toolbox of techniques catalogues development practices without giving any information about how they would fit into the development lifecycle [41].
- Foote's Big Ball of Mud pattern language can be read as advocacy, apologia, or critique of unorganised development, a defence of postmodern programming, or a parody of patterns in general [26]
- Perl [72]
- Usage-Centered Design deconstructs the subjects of our programs into many individual user rôles, and the intentions (desires) of those subjects into very many essential use cases [13].
- Groves has demonstrated how the refinement calculus can describe maintenance, modification, and refactoring, opening the way for *Extreme Formalism* [39].
- Scripting Languages (e.g. Tcl, Ruby, JavaScript) are designed to be only some part of the program, and are never written in themselves [56].
- *Lisp: Good News, Bad News, How to Win Big* offers a last-ditch defence of late modernism: *"I think there will be a next Lisp. This Lisp must be carefully designed, using the principles for success we saw in worse-is-better."* [29].
- Dream Machines and Computer Lib present the Digital Equipment Corp. line of computers in high postmodern style (a large-format comic-book) [53].
- Jackson's *Software Requirements & Specifications* provides a (postmodern, post-structured) dictionary of development terminology, while *Problem Frames* recontextualises his earlier Structured Design and Structured Programming Methods as one technique amongst many [42, 43].
- The Microsoft-brand Common Language Runtime supports many different programming languages with a common (late-modern) execution framework [36].
- The *Rational Software Process — How and Why to Fake it* describes why unifying irrational processes is more rational than Unified Processes [58, 48].
- Aspect- and Subject- Oriented Programming in Aspect/J, Hyper/J, and Composition Filters breaks down the recursive structure of object-oriented programs to introduce a multidimensional subdivision [45, 40, 1]. The Aspect Browser visualises the topics of discourse within program texts [38].
- Literate programming's continuation code sections deconstruct the rigorous recursive structure of Pascal programs so that parts of procedures can be presented in a order that suits explanation and documentation [46].
- Intentional Programming separates syntax and semantics, deconstructing languages so programs can be written in any language or style [16].
- Evolutionary computation, neural networks, and cellular automata reject large scale descriptions in favour of local action.
- *A Small Matter of Programming* [52].
- Anything to do with Spreadsheets [57, 63]
- Agile Methodologies and Extreme Programming. Development proceeds incrementally, customised to suit the occupational culture [12]. Note that Extreme Programming still insists on a grand narrative, in contrast to Agile development generally [27].
- Open Source and Mob Software development replace centralised development by a single company with mongolian hordes of programmers giving their time free across the internet [32, 60].
- Mr Bunny's Big Cup O' Java — Farmer Jake gets to places Niklaus Wirth can only dream about [23].
- Minimal Manuals — practical, descriptive information written on cards, with no precise ordering or overarching theory of operation [10].

15 Small Stories of Devotion

. . . We have the whole world
in our hands, smiles one. What the hell
are we going to do with it, laughs the other.

Small Stories of Devotion
Dinah Hawken

Gamma, Helm, Johnson and Vlissides *Design Patterns* remains for us, at least, a crucial text. It's not possible to give a single date for the start of postmodernism in computer science — as with other postmodernisms, "it seems to have slunk over the horizon" [64, p.158]. If pressed, we would choose OOPSLA 1994, where the *Design Patterns* book first became openly available. In a lovely postmodern irony, the book is copyrighted 1995.

Reflecting upon *Design Patterns* from the distance of the best part of a decade (an octade?) we should pause, once again to be surprised at its continued successes, not irritated over its failures. *Design Patterns* has sparked a number of imitators: many mediocre, some less so, none as successful as the original, along with several edited collections and a multinational multiannual conference metaseries. The idea of an object-oriented design pattern, of the kind described in the book, is now accepted throughout computer science practice, incorporated into the libraries and documentation for emerging programming languages such as Java and C♯, and taught routinely in most undergraduate programming curricula. The breadth of the authors' vision is clear that in the last eight years, although many patterns have been written on a variety of topics, less than ten additional object-oriented design patterns have been found that are of a piece with the original twenty-three.

There have been a number of more-or-less organised critiques of *Design Patterns*, arguing that the patterns approach betrayed the future of modern computer science (a conclusion with which we agree). The nature of this betrayal varies, of course: some arguing that patterns remove or resist formalisation, taking us to hell in a phenomenological handbasket [70, 22]; others that design patterns have corrupted the Alexanderian heritage of a pattern language into which all the patterns must fit [25, 30].

We contend that *Design Patterns* is postmodern precisely because it does not fit into an overarching prescriptive narrative of design: programmers are free to use or not use patterns as they see fit, as one of many techniques at their disposal. This makes it easy to adopt design patterns whatever personal or corporate philosophy you espouse. Precisely because patterns are small independent narratives, supported by arguments made on a case-by-case basis in favour of certain designs, it is easy to learn patterns piecemeal. The structure of the book is essentially arbitrary, although there are a number of distinct and subtle relationships between individual patterns [54]. *Design Patterns* certainly builds on modern techniques (cohesion and coupling, modern languages such as C++ or Smalltalk; OMT design notation): but this is not problematic — modern technology often ends up in the service of postmodern aesthetics.

Finally, we consider *Design Patterns* to be postmodern because it is concerned with the practice of programmers working out their own designs, embodied within the programs that they create. The focus of the book is the artifacts themselves: programs, designs, code, treated as objects of intellectual study and critique. But suffused all through the text, amid the concerns for pedagogy, efficiency, flexibility, and convincing argument, is the authors' clear respect for the topic of their discourses: their love of programs and programming.

> I think of the postmodern attitude as that of a man who loves a very cultivated woman and knows that he cannot say to her, 'I love you madly', because he knows that she knows (and that she knows that he knows) that these words have already been written by Barbara Cartland. Still, there is a solution. He can say, 'As Barbara Cartland would put it, I love you madly.' At this point, having avoided false innocence, having said clearly that it is no longer possible to speak innocently, he will nevertheless have said what he wanted to say to the woman: that he loves her, but he loves her in an age of lost innocence. If the woman goes along with this, she will have received a declaration of love all the same.

Reflections on the Name of the Rose [20]
Umberto Eco, 1985

References

[1] M. Aksit and A. Tripathi. Data abstraction mechanisms in SINA/ST. In *OOPSLA Proceedings*, 1988.

[2] Peter Bøgh Andersen. Computer semiotics. *Scandinavian Journal of Information systems*, 4:3–30, 1992.

[3] Peter Bøgh Andersen. *A Theory of Computer Semiotics*. Cambridge University Press, second edition, 1997.

[4] Peter Bøgh Andersen and Palle Nowack. Tangible objects — connecting informational and physical space. In Lars Qvortrup et al., editor, *Virtual Space: The Spatiality of Virtual Inhabitated 3D Worlds*. Springer-Verlag, 2001.

[5] Tony Becher and Paul Trowler. *Academic Tribes and Territories: Intellectual Enquiry and the Cultures of Discipline*. Open University Press, 2nd edition, 2001.

[6] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[7] Grady Booch. *Software Engineering with Ada*. Benjamin Cummings, 1983.

[8] Grady Booch. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.

[9] John Cage. *Silence*. Wesleyan Univ Press, 1973.

[10] John Carroll. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. MIT Press, 1990.

[11] Patrick Chan and Carleton Egremont III. *Mr Bunny's Internet Startup game*. Addison-Wesley, 2000.

[12] Larry L. Constantine. *The Peopleware Papers*. Prentice-Hall, 2001.

[13] Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley, 1998.

[14] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

[15] Ward Cunningham. WikiWikiWeb. `http://c2.com/cgi/wiki`.

[16] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[17] Edsger W. Dijkstra. Notes on structured programming. In Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[18] Edsger W. Dijkstra. How do we tell truths that might hurt? published as [19], June 1975.

[19] Edsger W. Dijkstra. How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective*, pages 129–131. Springer-Verlag, 1982.

[20] Umberto Eco. *Reflections on the Name of the Rose*. Secker & Warburg, 1985.

[21] Umberto Eco. *Semiotics and the Philosophy of Language*. Indiana University Press, 1986.

[22] A. H. Eden, A. Yehudai, and G. Gil. Precise specification and automatic application of design patterns. In *1997 International Conference on Automated Software Engineering (ASE'97)*, 1997.

[23] Carlton Egremont III. *Mr. Bunny's Big Cup o' Java*. Addison-Wesley, 1999.

[24] Richard Phillips Feynman. *What Do You Care What Other People Think?: Further Adventures of a Curious Character*. W.W. Norton, 1988.

[25] Brian Foote. The show trial of the gang of four for crimes against computer science. Panel at OOPSLA 1999. `http://www.laputan.org/patterns/gang-of-four.html`, November 1999.

[26] Brian Foote and Joe Yoder. Big ball of mud. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design*, volume 4, chapter 29, pages 653–692. Addison-Wesley, 2000.

[27] Martin Fowler. The new methodology. `http://www.martinfowler.com/articles/newMethodology.html`, November 2001.

[28] David Futrelle. Enron contra. *CNN MONEY*, January 2002. Jan 25th. `http://money.cnn.com/2002/01/25/techinvestor/futrelle`.

[29] Richard P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, 1991.

[30] Richard P. Gabriel. Back to the future: Worse (still) is better! `http://dreamsongs.com/NewFiles/ProWorseIsBetterPosition.pdf`, December 2000.

[31] Richard P. Gabriel. Wither software. `http://www.dreamsongs.com/NewFiles/WhitherSoftware.pdf`, March 2002.

[32] Richard P. Gabriel and Ron Goldman. *Mob Software: The Erotic Life of Code*. Dreamsongs Press, 2000.

[33] Richard P. Gabriel and Dave Thomas. Computing rainbow. Report of Feyerabend Workshop, `http://www.dreamsongs.com/Feyerabend/FeyerabendW4.html`, May 2001.

[34] William Gibson. *Neuromancer*. Ace Books, 1984.

[35] Les Goldschlager and Andrew Lister. *Computer Science : A Modern Introduction*. Prentice-Hall, 1982.

[36] John Gough. *Compiling for the .NET Common Language Runtime*. PTR PH, 2002.

[37] Ian Graham, Brian Henderson-Sellers, and Houman Younessi. *The OPEN Process Specification*. Addison-Wesley, 1997.

[38] William G. Griswold. The aspect browser. `http://www.cs.ucsd.edu/users/wgg/Software/AB/`, March 2002.

[39] Lindsay Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[40] Willian Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA Proceedings*, pages 411–428, 1993.

[41] Brian Henderson-Sellers, Anthony Simons, and Houman Younessi. *The OPEN Toolbox of Techniques*. Addison-Wesley, 1998.

[42] Michael Jackson. *Software Requirements & Specifications*. ACM, 1995.

[43] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[44] Charles Jencks. *The Language of Post-Modern Architecture*. Academy Editions, 1987.

[45] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP Proceedings*, pages 327–353, 2001.

[46] Donald Ervin Knuth. *Literate Programming*. CSLI Publications, 1992.

[47] Andrew Koenig. Idiomatic design. In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications*, pages 14–19. ACM Press, 1995.

[48] Philipe Krutchen. *The Rational Unified Process*. Addison-Wesley, 1999.

[49] B. Leuf and W. Cunningham. *The Wiki Way*. Addison-Wesley Publication Co., 2001.

[50] Jean-François Lyotard. From the postmodern condition. In Anthony Easthope and Kate McGowan, editors, *A Critical And Cultural Reader*. Allen & Unwin, 1992.

[51] Julie Mason. Auditing software raised 'red alert'. *Houston Chronicle*, January 2002. Jan 25th. `http://www.chron.com/cs/CDA/printstory.hts/business/1227548`.

[52] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.

[53] Theodor H. Nelson. *Computer Lib/Dream Machines*. Aperture, 1974.

[54] James Noble and Robert Biddle. Patterns as signs. In *ECOOP Proceedings*, 2002.

[55] James Noble, Robert Biddle, and Ewan Tempero. Metaphor and metonymy in object-oriented design patterns. In *Proceedings of Australian Computer Science Conference (ACSC)*. Australian Computer Society, 2002.

[56] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[57] Raymond D. Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10(2):15–21, Spring 1998.

[58] David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[59] Alan Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9), September 1982.

[60] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, 2001.

[61] H. Robinson, P. Hall, F. Hovenden, and J. Rachel. Postmodern software development. *The Computer Journal*, 31:363–375, 1998.

[62] Margaret Rose. *The Post-Modern and the Post-Industrial : A Critical Analysis*. Cambridge University Press, 1991.

[63] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. Wysiwyt testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239, June 2000.

[64] Stuart Sim, editor. *The Routledge Companion to Postmodernism*. Routledge, 2001.

[65] Alan Sokal and Jean Bricmont. *Intellectual Impostures*. Profile Books, July 1998.

[66] Jack B. Sowards. Star trek II: the Wrath of Kahn. Motion Picture.

[67] Bjarne Stroustrup. Why C++ is not just an object-oriented programming language. In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications*, pages 1–13. ACM Press, 1995.

[68] Antero Taivalsaari. Classes vs. prototypes: Some philosophical and historical observations. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming: Concepts, Languages and Applications*, chapter 1. Springer-Verlag, 1999.

[69] James B. Twitchell. *twenty ADS that shook the WORLD*. Three Rivers Press, 2000.

[70] Peter van Emde Boas. Resistance is futile; formal linguistic observations on design patterns. Technical report, University of Amsterdam, 1997.

[71] Andy Wachowski and Larry Wachowski. The Matrix. Technicolor 35mm Motion Picture, 1999.

[72] Larry Wall. Perl, the first postmodern computer language. `http://www.wall.org/~larry/pm.html`, Spring 1999.

[73] Nigel Wheale, editor. *The Postmodern Arts : An Introductory Reader*. Routledge, 1995.

[74] Winfried Wilcke. Computer architecture challenges in the next ten years. Keynote talk to Australian Computet Science Week 2002.

[75] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[77] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*. Routledge and Kegan Paul, 1961.

[78] Edward Yourdon and Jennifer Yourdon. *Time Bomb 2000!: What the Year 2000 Computer Crisis Means to You!* Prentice Hall PTR, 1997.

[79] Liping Zhao and James O. Coplien. Symmetry in class and type hierarchy. In James Noble and John Potter, editors, *In Proc. Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Conferences in Research and Practice in Information Technology. Australian Computer Society, 2002.

Principles of Lean Thinking

Mary Poppendieck
Poppendieck.LLC
7666 Carnelian Lane
Eden Prairie, MN 55346 USA
952-934-7998
mary@poppendieck.com

Abstract

In the 1980's, a massive paradigm shift hit factories throughout the US and Europe. Mass production and scientific management techniques from the early 1900's were questioned as Japanese manufacturing companies demonstrated that 'Just-in-Time' was a better paradigm. The widely adopted Japanese manufacturing concepts came to be known as 'lean production'. In time, the abstractions behind lean production spread to logistics, and from there to the military, to construction, and to the service industry. As it turns out, principles of lean thinking are universal and have been applied successfully across many disciplines.

Lean principles have proven not only to be universal, but to be universally successful at improving results. When appropriately applied, lean thinking is a well-understood and well-tested platform upon which to build agile software development practices.

Introduction

Call a doctor for a routine appointment and chances are it will be scheduled a few weeks later. But one large HMO in Minnesota schedules almost all patients within a day or two of their call, for just about any kind of medical service. A while ago, this HMO decided to worked off their schedule backlogs by extending their hours, and then vary their hours slightly from week to week to keep the backlog to about a day. True, the doctors don't have the comforting weeks-long list of scheduled patients, but in fact, they see just as many patients for the same reasons as they did before. The patients are much happier, and doctors detect medical problems far earlier than they used to.

The idea of delivering packages overnight was novel when Federal Express was started in 1971. In 1983, a new company called Lens Crafters changed the basis of competition in the eyeglasses industry by assembling prescription glasses in an hour. The concept of shipping products the same day they were ordered was a breakthrough concept when LL Bean upgraded its distribution system in the late 1980's. Southwest Airlines, one of the few profitable airlines these days, saves a lot of money with its unorthodox method of assigning seats as people arrive at the airport. Dell maintains profitability in a cutthroat market by manufacturing to order in less than a week. Another Austin company builds custom homes in 30 days.

The common denominator behind these and many other industry-rattling success stories is lean thinking. Lean thinking looks at the value chain and asks: How can things be structured so that the enterprise does nothing but add value, and does that as rapidly as possible? All the intermediate steps, all the intermediate time and all the intermediate people are eliminated. All that's left are the time, the people and the activities that add value for the customer.

Origins of Lean Thinking

Lean thinking got its name from a 1990's best seller called *The Machine That Changed the World : The Story of Lean Production*[1]. This book chronicles the movement of automobile manufacturing from craft production to mass production to lean production. It tells the story of how Henry Ford standardized automobile parts and assembly techniques, so that low skilled workers and specialized machines could make cheap cars for the masses. The book goes on to describe how mass production provided cheaper cars than the craft production, but resulted an explosion of indirect labor: production planning, engineering, and management. Then the book explains how a small company set its sights set on manufacturing cars for Japan, but it could not afford the enormous investment in single purpose machines that seemed to be required.

[1] The Machine That Changed the World : The Story of Lean Production, by Womack, James P., Daniel T. Jones, and Daniel Roos, New York: Rawson and Associates; 1990.

Nor could it afford the inventory or large amount of indirect labor that seemed necessary for mass production. So it invented a better way to do things, using very low inventory and moving decision-making to production workers. Now this small company has grown into a large company, and the Toyota Production System has become known as 'lean production'.

"The mass-producer uses narrowly skilled professionals to design products make by unskilled or semiskilled workers tending expensive, single-purpose machines. These churn out standardized products at high volume. Because the machinery costs so much and is so intolerant of disruption, the mass-producer adds many buffers – extra supplies, extra workers, and extra space – to assure smooth production…. The result: The customer gets lower costs but at the expense of variety and by means of work methods that most employees find boring and dispiriting."[2]

Think of the centralized eyeglasses laboratory. Remember that Sears used to take two or three weeks to fill orders from its once-popular catalog. Recall the long distribution channel that used to be standard in the computer market. Think dinosaurs. Centralized equipment, huge distribution centers and lengthy distribution channels were created to realize economies of scale. They are the side effects of mass-production, passed on to other industries. What people tend to overlook is that mass-production creates a tremendous amount of work that does not directly add value. Shipping eyeglasses to a factory for one hour of processing adds more handling time by far than the processing time to make the glasses. Adding retail distribution to the cutthroat personal computer industry means that a manufacturer needs six weeks to respond to changing technology, instead of six days. Sears' practice of building an inventory of mail orders to fill meant keeping track of stacks of orders, not to mention responding to innumerable order status queries and constant order changes.

"The lean producer, by contrast, combines the advantages of craft and mass production, while avoiding the high cost of the former and the rigidity of the later… Lean production is 'lean' because it uses less of everything compared with mass production – half the human effort in the factory, half the manufacturing space, half the investment in tools, half the engineering hours to develop a new product in half the time. Also, it requires keeping far less than half the inventory on site, results in many fewer defects, and produces a greater and ever growing variety of products."[3]

While on a tour of a large customer, Michael Dell saw technicians customizing new Dell computers with their company's 'standard' hardware and software. "Do you think you guys could do this for me?" his host asked. Without missing a beat, Dell replied, "Absolutely, we'd love to do that."[4] Within a couple of weeks, Dell was shipping computers with factory-installed, customer-specific hardware and software. What took the customer an hour could be done in the factory in minutes, and furthermore, computers could be shipped directly to end-users rather than making a stop in the corporate IT department. This shortening of the value chain is the essence of lean thinking.

Companies that re-think the value chain and find ways to provide what their customers value with significantly fewer resources than their competitors can develop an unassailable competitive advantage. Sometimes competitors are simply not able to deliver the new value proposition. (Many have tired to copy Dell; few have succeeded.) Sometimes competitors do not care to copy a new concept. (Southwest Airlines has not changed the industry's approach to seat assignments.) Sometimes the industry follows the leader, but it takes time. (Almost all direct merchandise is shipped within a day or two of receiving an order these days, but the Sears catalog has been discontinued.)

Lean Thinking in Software Development

eBay is a company which pretty much invented 'lean' trading by eliminating all the unnecessary steps in the trading value chain. In the mid 1990's, basic eBay software capabilities were developed by responding daily to customer requests for improvements.[5] Customers would send an e-mail to Pierre Omidyar with a suggestion and he would implement the idea on the site that night. The most popular features of eBay, those which create the highest competitive advantage, were created in this manner.

Digital River invented the software download market in the mid 1990's by focusing on 'lean' software delivery. Today Digital River routinely designs and deploys

[2] Womack (1990) p 13.

[3] Womack (1990) p 13.

[4] _Direct from Dell_, by Michael Dell with Catherine Fredman, Harper Business, 1999, p 159

[5] _Q&A with eBay's Pierre Omidyar,_ Business Week Online, December 3, 2001.

sophisticated web sites for corporate customers in a matter of a weeks, by tying the corporation's legacy databases to standard front end components customized with a 'look and feel' specific to each customer.

In the mid 1990's, Microsoft implemented corporate-wide financial, purchasing and human resource packages linked to data warehouses which can be accessed via web front-ends. Each was implemented by "a handful of seasoned IT and functional experts… (who got) the job done in the time it takes a … committee to decide on its goals."[6]

In each of these examples, the focus of software development was on rapid response to an identified need. Mechanisms were put in place to dramatically shorten the time from problem recognition to software solution. You might call it 'Just-in-Time' software development.

The question is – why isn't all software developed quickly? The answer is – rapid development must be considered important before it becomes a reality. Once speed becomes a value, a paradigm shift has to take place, changing software development practices from the mass production paradigm to lean thinking.

If your company writes reams of requirements documents (equivalent to inventory), spends hours upon hours tracking change control (equivalent to order tracking), and has an office which defines and monitors the software development process (equivalent to industrial engineering), you are operating with mass-production paradigms. Think 'lean' and you will find a better way.

Basic Principles of Lean Development

There are four basic principles of lean thinking which are most relevant to software development:

The Basic Principles of Lean Development
Add Nothing But Value (Eliminate Waste)
Center On The People Who Add Value
Flow Value From Demand (Delay Commitment)
Optimize Across Organizations

[6] *Inside Microsoft: Balancing Creativity and Discipline*, Herbold, Robert J.; *Harvard Business Review,* January 2002.

Add Nothing But Value (Eliminate Waste)

The first step in lean thinking is to understand what value is and what activities and resources are absolutely necessary to create that value. Once this is understood, everything else is waste. Since no one wants to consider what they do as waste, the job of determining what value is and what adds value is something that needs to be done at a fairly high level. Let's say you are developing order tracking software. It seems like it would be very important for a customer to know the status of their order, so this would certainly add customer value. But actually, if the order is in house for less than 24 hours, the only order status that is necessary is to inform the customer that the order was received, and then that it has shipped, and let them know the shipping tracking number. Better yet, if the order can be fulfilled by downloading it on the Web, there really isn't any order status necessary at all.

To develop breakthroughs with lean thinking, the first step is learning to see waste. If something does not directly add value, it is waste. If there is a way to do without it, it is waste. Taiichi Ohno, the mastermind of the Toyota Production System, identified seven types of manufacturing waste:

The Seven Wastes of Manufacturing
Overproduction
Inventory
Extra Processing Steps
Motion
Defects
Waiting
Transportation

Here is how I would translate the seven wastes of manufacturing to software development:

The Seven Wastes of Software Development
Overproduction = Extra Features
Inventory = Requirements
Extra Processing Steps = Extra Steps
Motion = Finding Information
Defects = Defects Not Caught by Tests
Waiting = Waiting, Including Customers
Transportation = Handoffs

Extreme Programming (XP) is a set of practices which focuses on rapid software development. It is interesting to examine how XP works to eliminate the seven wastes of software development:

Waste in Software Development	**How Extreme Programming Addresses Waste**
Extra Features	Develop only for today's stories
Requirements	Story cards are detailed only for the current iteration
Extra Steps	Code directly from stories; get verbal clarification directly from customers
Finding Information	Have everyone in the same room; customer included
Defects Not Caught by Tests	Test first; both developer tests and customer tests
Waiting, Including Customers	Deliver in small increments
Handoffs	Developers work directly with customers

'Do It Right The First Time'

XP advocates developing software for the current need, and as more 'stories' (requirements) are added, the design should be 'refactored'[7] to accommodate the new stories. Is it waste to refactor software? Shouldn't developers "Do It Right the First Time?"

It is instructive to explore the origins of the slogan "Do It Right the First Time." In the 1980's it was very difficult to change a mass-production plant to lean production, because in mass production, workers were not expected to take responsibility for the quality of the product. To change this, the management structure of the plant had to change. "Workers respond only when there exists some sense of reciprocal obligation, a sense that management actually values skilled workers, … and is willing to delegate responsibility to [them]."[8] The slogan "Do It Right the First Time" encouraged workers to feel responsible for the products moving down the line, and encourage them to stop the line and troubleshoot problems when and where they occurred.

[7] Refactoring is improving the design of software without changing functionality.

[8] Womack (1990) p 99.

In the software industry, the same slogan "Do It Right the First Time," has been misused as an excuse to apply mass-production thinking, not lean thinking to software development. Under this slogan, ***responsibility has been taken away from the developers who add value***, which is exactly the opposite of its intended effect. "Do It Right the First Time" has been used as an excuse to insert reams of paperwork and armies of analysts and designers between the customer and the developer. In fact, the slogan is only properly applied if it gives developers more, not less, involvement in the results of their work.

A more appropriate translation of such slogans as "Zero Defects" and "Do It Right the First Time" would be "Test First". In other words, don't code unless you understand what the code is supposed to do and have a way to determine whether the code works. A good knowledge of the domain coupled with short build cycles and automated testing constitute the proper way for software developers to "Do It Right the First Time".

Center On The People Who Add Value

Almost every organization claims it's people are important, but if they truly center on those who add value, they would be able to say:

The people doing the work are the center of
Resources
Information
Process Design Authority
Decision Making Authority
Organizational Energy

In mass-production, tasks are structured so that low skilled or unskilled workers can easily do the repetitive work, but engineers and managers are responsible for production. Workers are not allowed to modify or stop the line, because the focus is to maintain volume. One of the results of mass-production is that unskilled workers have no incentive to volunteer information about problems with the manufacturing line or ways to improve the process. Maladjusted parts get fixed at the end of the line; a poor die or improperly maintained tool is management's problem. Workers are neither trained nor encouraged to worry about such things.

"The truly lean plant has two key organizational features: *It transfers the maximum number of tasks and responsibilities to those workers actually adding value to the car on the line, and it has in place a system for* *detecting defects that quickly traces every problem, once discovered, to its ultimate cause."*[9] Similarly in any lean enterprise, the focus is on the people who add value. In lean enterprises, traditional organizational structures give way to new team-oriented organizations which are centered on the flow of value, not on functional expertise.

The first experiment Taiichi Ohno undertook in developing lean production was to figure out a way to allow massive, single-purpose stamping machines to stamp out multiple parts. Formerly, it took skilled machinists hours, if not days, to change dies from one part to another. Therefore, mass production plants had many single purpose stamping machines in which the dies were almost never changed. Volume, space, and financing were not available in Japan to support such massive machines, so Ohno set about devising simple methods to change the stamping dies in minutes instead of hours. This would allow many parts of a car to be made on the same line with the same equipment. Since the workers had nothing else to do while the die was being changed, they also did the die changing, and in fact, the stamping room workers were involved in developing the methods of rapid die changeover.

Ohno transferred most of the work being done by engineers and managers in mass-production plants to the production workers. He grouped workers in small teams and trained the teams to do their own industrial engineering. Workers were encouraged to stop the line if anything went wrong, (a management job in mass-production). Before the line was re-started, the workers were expected to search for the root cause of the problem and resolve it. At first the line was stopped often, which would have been a disaster at a mass-production plant. But eventually the line ran with very few problems, because the assembly workers felt responsible to find, expose, and resolve problems as they occurred.

It is sometimes thought that a benefit of good software engineering is to allow low skilled programmers to produce code while a few high skilled architects and designers do the critical thinking. With this in mind, a project is often divided into requirements gathering, analysis, design, coding, testing, and so on, with decreasing skill presumably required at each step. A 'standard process' is developed for each step, so that low-skilled programmers, for example, can translate design into code simply by following the process.

This kind of thinking comes from mass-production, where skilled industrial engineers are expected to design production work for unskilled laborers. It is the antithesis of lean thinking and devalues the skills of the developers who actually write the code as surely as industrial engineers telling laborers how to do their jobs devalues the skills of production workers.

Centering on the people who add value means upgrading the skills of developers through training and apprenticeships. It means forming teams that design their own processes and address complete problems. It means that staff groups and managers exist to support developers, not to tell them what to do.

Flow Value From Demand (Delay Commitment)

The idea of flow is fundamental to lean production. If you do nothing but add value, then you should add the value in as rapid a flow as possible. If this is not the case, then waste builds up in the form of inventory or transportation or extra steps or wasted motion. The idea that flow should be 'pulled' from demand is also fundamental to lean production. 'Pull' means that nothing is done unless and until an upstream process requires it. The effect of 'pull' is that production is not based on forecast; commitment is delayed until demand is present to indicate what the customer really wants.

Pulling from demand can be one of the easiest ways to implement lean principles, as LL Bean and Lens Crafters and Dell found out. The idea is to fill each customer order immediately. In mass-production days, filling orders immediately meant building up lots of inventory in anticipation of customer orders. Lean production changes that. The idea is to be able to make the product so fast that it can be made to order. True, Dell and Lens Crafters and LL Bean and Toyota have to have some inventory of sub-assemblies waiting to be turned into a finished product at a moments notice. But it's amazing how little inventory is necessary, if the process to replenish the inventory is also lean. A truly lean distribution channel only works with a really lean supply chain coupled to very lean manufacturing.

The "batch and queue" habit is very hard to break. It seems counterintuitive that doing a little bit at a time at the last possible moment will give faster, better, cheaper results. But anyone designing a control system knows that a short feedback loop is far more effective at maintaining control of a process than a long loop. The problem with batches and queues is that they hide problems. The idea of lean production is to expose

[9] Womack (1990) p 99. Italics in the original.

problems as soon as they arise, so they can be corrected immediately. It may seem that lean systems are fragile, because they have no padding. But in fact, lean systems are quite robust, because they don't hide unknown, lurking problems and they don't pretend they can forecast the future.

In Lean Software Development, the idea is to maximize the flow of information and delivered value. As in lean production, maximizing flow does not mean automation. Instead, it means limiting what has to be transferred, and transferring that as few times as possible over the shortest distance with the widest communication bandwidth as late as is possible. Handing off reams of frozen documentation from one function to the next is a mass-production mentality. In Lean Software Development, the idea is to eliminate as many documents and handoffs as possible. Documents which are not useful to the customer are replaced with automated tests. These tests assure that customer value is delivered both initially and in the future when the inevitable changes are needed.

In addition to rapid, Just-in-Time information flow, Lean Software Development means rapid, Just-in-Time delivery of value. In manufacturing, the key to achieving rapid delivery is to manufacture in small batches pulled by a customer order. Similarly in software development, the key to rapid delivery is to divide the problem into small batches (increments) pulled by a customer story and customer test. The single most effective mechanism for implementing lean production is adopting Just-in-Time, pull-from-demand flow. ***Similarly, the single most effective mechanism for implementing Lean Development is delivering increments of real business value in short time-boxes.***

In Lean Software Development, the goal is to eliminate as many documents and handoffs as possible. The emphasis is to pair a skilled development team with a skilled customer team and give them the responsibility and authority to develop the system in small, rapid increments, driven by customer priority and feedback.

Optimize across Organizations

Quite often, the biggest barrier to adopting lean practices is organizational. As products move from one department to another, a big gap often develops, especially if each department has its own set of performance measurements that are unrelated to the performance measurements of neighboring departments.

For example, let's say that the ultimate performance measurement of a stamping room is machine productivity. This measurement motivates the stamping room to build up mounds of inventory to keep the machines running at top productivity. It does not matter that the inventory has been shown to degrade the overall performance of the organization. As long as the stamping room is measured primarily on machine productivity, it will build inventory. This is what is known as a sub-optimizing measurement, because it creates behavior which creates local optimization at the expense of overall optimization.

Sub-optimizing measurements are very common, and overall optimization is virtually impossible when they are in place. One of the biggest sub-optimizing measurements in software development occurs when project managers measured on earned value. Earned value is the cost initially estimated for the tasks which have been completed. The idea is that you had better not have spent any more than you estimated. The problem is, this requires a project manager to build up an inventory of task descriptions and estimates. Just as excess inventory in the stamping room slows down production and degrades over time, the inventory of tasks required for earned value calculations gets in the way of delivering true business value and also degrades over time. Nevertheless, if there is an earned value measurement in place, project tasks are specified and estimated, and earned value is measured. When it comes to a choice between delivering business value or earned value (and it often does), earned value usually wins out.

To avoid these problems, lean organizations are usually structured around teams that maintain responsibility for overall business value, rather than intermediate measurements such as their ability to speculate and pad estimates. Another approach is to foster a keen awareness that the downstream department is a customer, and satisfying this internal customer is the ultimate performance measurement.

The paradigm shift that is required with lean thinking is often hindered if the organization is not structured around the flow of value and focused on helping the customer pull value from the enterprise. For this reason, software development teams are best structured around delivering increments of business value, with all the necessary skills on the same team (eg. customer understanding / domain knowledge, architecture / design, system development, database administration, testing, system administration, etc.).

Software Development Contracts

Flow along the value stream is particularly difficult when multiple companies are involved. Many times I have heard the lament: "Everything you say makes sense, but it is impossible to implement in our environment, because we work under contracts with other organizations." Indeed, the typical software development contract can be the ultimate sub-optimizing mechanism. Standard software contracts and supplier management practices have a tendency to interfere with many lean principles.

Manufacturing organizations used to have the same problem. For example, US automotive companies once believed the best way to reduce the cost of parts in an automobile was with annual competitive bidding. If the only thing that is important is cheap parts, competitive bidding may seem like the best way to achieve this goal. However, if overall company performance is more important, then better parts which integrate more effectively with the overall vehicle are more valuable. In fact, there is an direct correlation between an automotive company's profitability and its degree of collaboration with suppliers.[10] When Chrysler moved from opportunistic to collaborative relationships with its suppliers in the late 1990's, it's performance improved significantly.

The software industry has some lessons to learn in the area of contractual agreements between organizations. It needs to learn how to structure collaborative relationships which maximize the overall results of both parties. A key lesson the software industry needs to learn is how to structure contracts for incremental deliveries that are not pre-defined in the contract, yet assure the customer of prompt delivery of business value appropriate to their investment. Here again, we can learn from lean production.

Lean manufacturing organizations develop a limited number of relationships with 'trusted' suppliers, and in turn, gain the 'trust' of these suppliers. What does 'trust' mean? "Trust [is] one party's confidence that the other party in the exchange relationship will fulfill its promises and commitments and will not exploit its vulnerabilities."[11] "…trust…[is] not based on greater interpersonal trust, but rather greater trust in the fairness, stability, and predictability of [the company's] routines and processes."[12]

It has been the practice of legal departments writing software contracts to put into contractual language all of the protections necessary to keep the other side 'honest.' However, the transaction costs associated with creating and monitoring such contracts are enormous. Many contracts all but demand a waterfall process, even if both companies believe this is not the best approach. It's time that the software development industry learned the lesson of Supply Chain Management – "Extraordinary productivity gains in the production network or *value chain* are possible when companies are willing to collaborate in unique ways, often achieving competitive advantage by sharing resources, knowledge, and assets…. Today competition occurs between value chains and not simply between companies."[13]

Summary and Conclusion

The lean production metaphor is a good one for software development, if it is applied in keeping with the underlying spirit of lean thinking. In the past, the application of some manufacturing concepts to software development ('Do It Right the First Time' comes to mind) may have lacked a deep understanding of what makes lean principles work. The underlying principles of eliminating waste, empowering front line workers, responding immediately to customer requests, and optimizing across the value chain are fundamental to lean thinking. When applied to software development, these concepts provide a broad framework for improving software development.

[10] *Collaborative Advantage,* by Jeffrey H. Dyer, Oxford University Press; 2000, p 6.

[11] Dyer (2000) p 88.

[12] Dyer (2000) p 100

[13] Dyer (2000) p 5

Many-to-Many Invocation: A New Object Oriented Paradigm for Ad Hoc Collaborative Systems

Alan Kaminsky
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
ark@cs.rit.edu

Hans-Peter Bischof
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
hpb@cs.rit.edu

July 16, 2002

Abstract

Many-to-Many Invocation (M2MI) is a new paradigm for building collaborative systems that run in wireless proximal ad hoc networks of fixed and mobile computing devices. M2MI is useful for building a broad range of systems, including multiuser applications (conversations, groupware, multiplayer games); systems involving networked devices (printers, cameras, sensors); and collaborative middleware systems. M2MI provides an object oriented method call abstraction based on broadcasting. An M2MI invocation means "Every object out there that implements this interface, call this method." An M2MI-based application is built by defining one or more interfaces, creating objects that implement those interfaces in all the participating devices, and broadcasting method invocations to all the objects on all the devices. M2MI is layered on top of a new messaging protocol, the Many-to-Many Protocol (M2MP), which broadcasts messages to all nearby devices using the wireless network's inherent broadcast nature instead of routing messages from device to device. M2MI synthesizes remote method invocation proxies dynamically at run time, eliminating the need to compile and deploy proxies ahead of time. As a result, in an M2MI-based system, central servers are not required; network administration is not required; complicated, resource-consuming ad hoc routing protocols are not required; and system development and deployment are simplified.

1 Introduction

This paper describes a new paradigm, Many-to-Many Invocation (M2MI), for building collaborative systems that run in wireless proximal ad hoc networks of fixed and mobile computing devices. M2MI is useful for building a broad range of systems, including multiuser applications (conversations, groupware, multiplayer games); systems involving networked devices (printers, cameras); wireless sensor networks; and collaborative middleware systems.

M2MI provides an object oriented method call abstraction based on broadcasting. An M2MI-based application broadcasts method invocations, which are received and performed by many objects in many target devices simultaneously. An M2MI invocation means "Everyone out there that implements this interface, call this method." The calling application does not need to know the identities of the target devices ahead of time, does not need to explicitly discover the target devices, and does not need to set up individual connections to the target devices. The calling device simply broadcasts method invocations, and all objects in the proximal network that implement those methods will execute them.

As a result, M2MI offers these key advantages over existing systems:

- M2MI-based systems *do not require central servers;* instead, applications run collectively on the proximal devices themselves.
- M2MI-based systems *do not require network administration* to assign addresses to devices, set up routing, and so on, since method invocations are broadcast to all nearby devices. Consequently,

- M2MI is *well-suited for an ad hoc networking environment* where central servers may not be available and devices may come and go unpredictably.
- M2MI-based systems *do not need complicated ad hoc routing protocols* that consume memory, processing, and network bandwidth resources. Consequently,
- M2MI is *well-suited for small mobile devices* with limited resources and battery life.
- M2MI *simplifies system development* in several ways. By using M2MI's high-level method call abstraction, developers avoid having to work with low-level network messages. Since M2MI does not need to discover target devices explicitly or set up individual connections, developers need not write the code to do all that.
- M2MI *simplifies system deployment* by eliminating the need for always-on application servers, lookup services, codebase servers, and so on; by eliminating the software that would otherwise have to be installed on all these servers; and by eliminating the need for network configuration.

M2MI's key technical innovations are these:

- M2MI employs a *new message broadcasting protocol,* the Many-to-Many Protocol (M2MP), which uses a fundamentally different approach compared to existing ad hoc networking protocols. Instead of routing messages from point to point to the particular destination devices, M2MP broadcasts messages to all nearby devices, taking advantage of the wired or wireless network's inherent broadcast nature. Based on the message contents, the devices then decide whether and how to process the message.
- M2MI layers an *object oriented abstraction* on top of broadcast messaging, letting the application developer work with high-level method calls instead of low-level network messages.
- M2MI uses *dynamic proxy synthesis* to create remote method invocation proxies (stubs and skeletons) automatically at run time — as opposed to existing remote method invocation systems which compile the proxies offline and which must deploy the proxies ahead of time.

The paper is organized as follows. Section 2 describes the application domain and networking environment at which M2MI is targeted. Section 3 describes the M2MI paradigm at a conceptual level. Section 4 gives several examples of ad hoc collaborative systems based on M2MI. Sections 5, 6, and 7 describe the architecture and design of the M2MI software. Section 8 compares and contrasts M2MI with related work. Section 9 discusses the current status of M2MI and plans for further work.

2 Target Environment

M2MI's target domain is *ad hoc collaborative systems:* systems where multiple users with computing devices, as well as multiple standalone devices like printers, cameras, and sensors, all participate simultaneously *(collaborative);* and systems where the various devices come and go and so are not configured to know about each other ahead of time *(ad hoc).* Examples of ad hoc collaborative systems include:

- Multiuser applications: a chat session, a shared whiteboard, a group appointment scheduler, a file sharing application, or a multiplayer game.
- Applications that discover and use nearby networked services: a document printing application that finds printers wherever the user happens to be, or a surveillance application that displays images from nearby video cameras.
- Collaborative middleware systems like shared tuple spaces [14, 7].

In many such collaborative systems, every device needs to talk to every other device. Every person's chat messages are displayed on every person's device; every person's calendar on every person's device is queried and updated with the next meeting time. In contrast to applications like email or web browsing (one-to-one communication) or webcasting (one-to-many communication), the collaborative systems envisioned here exhibit *many-to-many communication patterns* (Figure 1). M2MI is designed especially to support applications with many-to-many communication patterns, although it also supports other communication patterns.

M2MI is also designed to take advantage of a wireless proximal ad hoc networking environment. The devices in the system connect to each other using *wireless* networking technology such as IEEE 802.11 or Bluetooth. The devices are located in *proximity* to each other, around the same table or in the same room; every device can hear every other device, Consequently, each transmitted message is immediately received by all the devices without needing to route the message through intermediate devices. Devices come and go as the system is running, and the devices

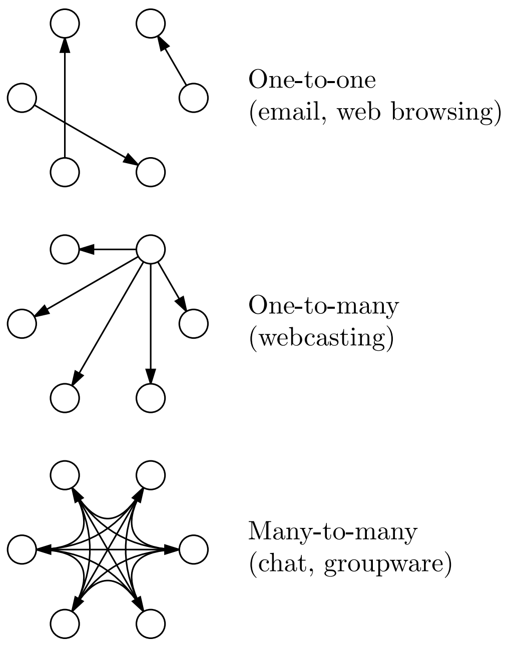


Figure 1: Communication patterns

do not know each others' identities beforehand; instead, the devices form *ad hoc* networks among themselves.

M2MI is intended for running collaborative systems *without central servers.* In a wireless ad hoc network of devices, relying on servers in a wired network is unattractive because the devices are not necessarily always in range of a wireless access point. Furthermore, relying on any one wireless device to act as a server is unattractive because devices may come and go without prior notification. Instead, all the devices — whichever ones happen to be present in the changing set of proximal devices — act in concert to run the system.

M2MI is intended to run in *small, battery powered* devices with limited memory sizes and CPU capacity. Unlike desktop computers, such devices cannot maintain constant network connections because that would rapidly drain their batteries. To make each battery charge last as long as possible, reducing network utilization is essential.

To reduce the amount of network traffic, M2MI takes advantage of the *broadcast* communication made possible by a wireless proximal network. In a collaborative system with n devices where every device sends messages to every other device, if messages had to be sent between individual devices, the number of messages would be proportional to n^2. But since M2MI uses broadcast messaging, the number of messages sent is only proportional to n. This also improves the scalability of M2MI, since the network traffic tends to increase linearly rather than quadratically as more devices join an M2MI-based system.

Although M2MI is designed to work well in a limited environment of small battery-powered devices, ad hoc wireless networks, and no central servers, M2MI is not confined to that environment. M2MI is perfectly capable of working in an environment of large host computers, wired networks, and central servers.

3 The M2MI Paradigm

Remote method invocation (RMI) [49] can be viewed as an object oriented abstraction of point-to-point communication: what looks like a method call is in fact a message sent and a response sent back. In the same way, M2MI can be viewed as an object oriented abstraction of broadcast communication. This section describes the M2MI paradigm at a conceptual level.

3.1 Handles

M2MI lets an application invoke a method declared in an interface. To do so, the application needs some kind of "reference" upon which to perform the invocation. In M2MI, a reference is called a *handle*, and there are three varieties, omnihandles, unihandles, and multihandles.

3.2 Omnihandles

An *omnihandle* for an interface stands for "every object out there that implements this interface." An application can ask the M2MI layer to create an omnihandle for a certain interface X, called the omnihandle's *target interface.* (A handle can implement more than one target interface if desired.) Figure 2 depicts an omnihandle for interface `Foo`; the omnihandle is named `allFoos`. It is created by code like this:

```
Foo allFoos = (Foo) M2MI.getOmnihandle
    (Foo.class);
```

Once an omnihandle is created, calling method Y on the omnihandle for interface X means, "Every object out there that implements interface X, perform method Y." The method is actually performed by whichever objects implementing interface X exist at the time the method is *invoked* on the omnihandle. Thus, different objects could respond to an omnihandle invocation at different times. Figure 3 shows what happens when the statement `allFoos.y();` is executed. Three objects implementing interface `Foo`,

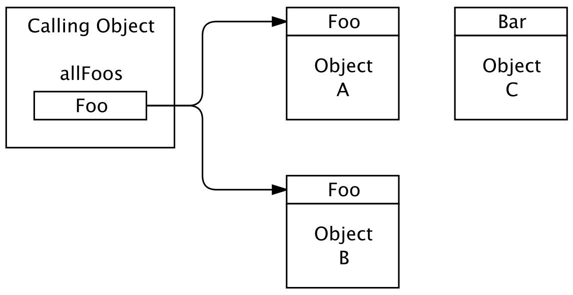


Figure 2: An omnihandle

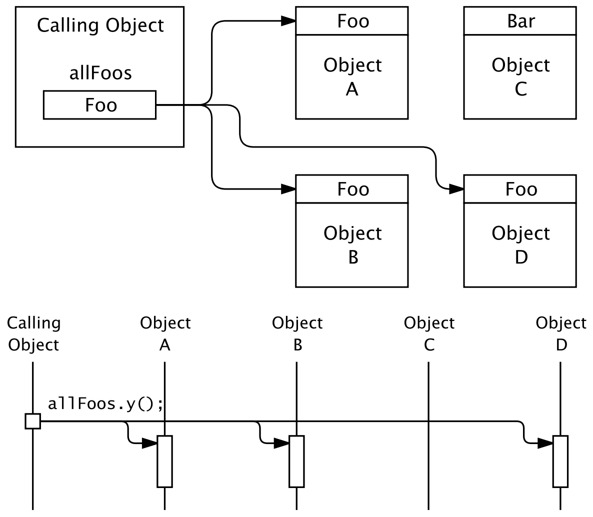


Figure 3: Invoking a method on an omnihandle

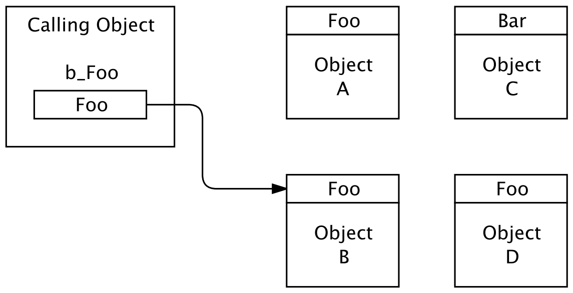


Figure 4: A unihandle

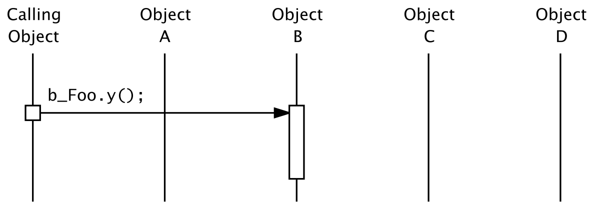


Figure 5: Invoking a method on a unihandle

objects A, B, and D, happen to be in existence at that time; so all three objects perform method `y`. Note that even though object D did not exist when the omnihandle `allFoos` was created, the method is nonetheless invoked on object D.

The target objects invoked by an M2MI method call need not reside in the same process as the calling object. The target objects can reside in other processes or other devices. As long as the target objects are in range to receive a broadcast from the calling object over the network, the M2MI layer will find the target objects and perform a *remote* method invocation on each one. (M2MI's remote method invocation does not, however, use the same mechanism as Java RMI.)

3.3 Exporting Objects

To receive invocations on a certain interface X, an application creates an object that implements interface X and *exports* the object to the M2MI layer. Thereafter, the M2MI layer will invoke that object's method Y whenever anyone calls method Y on an omnihandle for interface X. An object is exported with code like this:

```
  M2MI.export (b, Foo.class);
```

`Foo.class` is the class of the target interface through which M2MI invocations will come to the object. We say the object is "exported as type `Foo`." M2MI also lets an object be exported as more than one target interface.

Once exported, an object stays exported until explicitly unexported:

```
  M2MI.unexport (b);
```

In other words, M2MI does not do distributed garbage collection (DGC). In many distributed collaborative applications, DGC is unwanted; an object that is exported by one device as part of a distributed application should remain exported even if there are no other devices invoking the object yet. In cases where DGC is needed, it can be provided by a leasing mechanism [15, 1] explicit in the interface.

3.4 Unihandles

A *unihandle* for an interface stands for "one particular object out there that implements this interface." An application can export an object and have the M2MI layer return a unihandle for that object. Unlike an omnihandle, a unihandle is bound to one particular object at the time the unihandle is created. Figure 4 depicts a unihandle for object B implementing interface `Foo`; the unihandle is named `b_Foo`. It is created by code like this:

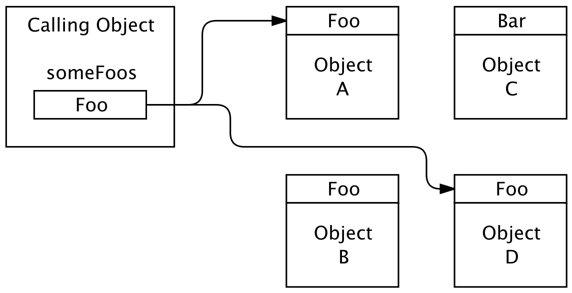


Figure 6: A multihandle

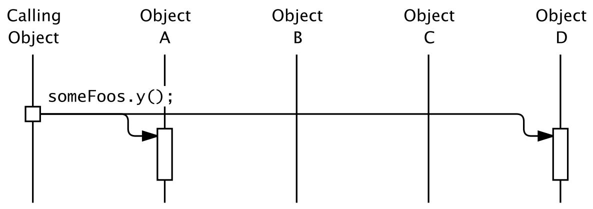


Figure 7: Invoking a method on a multihandle

```
Foo b_Foo = (Foo) M2MI.getUnihandle
   (b, Foo.class);
```

Once a unihandle is created, calling method Y on the unihandle means, "The particular object out there associated with this unihandle, perform method Y." When the statement `b_Foo.y();` is executed, only object B performs the method, as shown in Figure 5. As with an omnihandle, the target object for a unihandle invocation need not reside in the same process or device as the calling object.

A unihandle can be detached from its object, after which the object can no longer be invoked via the unihandle:

```
b_Foo.detach();
```

3.5 Multihandles

A *multihandle* for an interface stands for "one particular set of objects out there that implement this interface." Unlike a unihandle which only refers to one object, a multihandle can refer to zero or more objects. But unlike an omnihandle which automatically refers to all objects that implement a certain target interface, a multihandle only refers to those objects that have been explicitly *attached* to the multihandle. Figure 6 depicts a multihandle implementing target interface `Foo`; the multihandle is named `someFoos`, and it is attached to two objects, A and D. The multihandle is created and attached to the objects by code like this:

```
Foo someFoos = (Foo) M2MI.getMultihandle
   (Foo.class);
someFoos.attach (a);
someFoos.attach (d);
```

Once a multihandle is created, calling method Y on the multihandle means, "The particular object or objects out there associated with this multihandle, perform method Y." When the statement `someFoos.y();` is executed, objects A and D perform the method, but not objects B or C, as shown in Figure 7. As with an omnihandle or unihandle, the target objects for a multihandle invocation need not reside in the same process or device as the calling object or each other. A multihandle can be created in one process and sent to another process, and the destination process can then attach its own objects to the multihandle.

An object can also be detached from a multihandle:

```
someFoos.detach (a);
```

3.6 Characteristics of M2MI Invocations

Methods in interfaces invoked via M2MI can have arguments. When an object of a non-primitive type, including an array type, is passed directly as an M2MI method call argument, the object is normally *passed by copy;* manipulations of the argument by the method call recipient do not affect the original object in the caller. However, when a unihandle for an exported object is passed as an M2MI method call argument, the object is effectively *passed by reference;* invocations performed by the method call recipient on the argument (unihandle) come back to the original object via M2MI and thus do affect the original object in the caller. An omnihandle or multihandle can also be passed as an M2MI method call argument, and it behaves the same way in the method call recipient as it does in the caller. Primitive types are always passed by copy in M2MI.

M2MI uses Java's object serialization to marshal the method call arguments on the calling side and unmarshal them again on the target side. Accordingly, every non-primitive object passed in as an M2MI method call argument must be serializable, or the invocation will fail.

While M2MI can pass objects as arguments like Java RMI, M2MI does not download the argument objects' classes to the destination as Java RMI does. With M2MI, the destination must already possess the argument objects' classes, or the invocation will fail. If a handle is passed as an argument in an M2MI method call, though, the destination need only possess the handle's target interface or interfaces. (The destination's M2MI layer already possesses all the classes needed to implement handles.)

Although they can have arguments, methods in interfaces invoked via M2MI must be declared not to return a value and not to throw any exceptions. This is because with potentially more than one object performing the method, there is no single return value or exception to return or throw.

Since an M2MI method does not return anything, the caller cannot get any information back from the called object *in the same method call.* If the caller needs to get information back, the caller can send a reference to its own object by passing the object's unihandle as an argument to a method invoked on a handle. The called object or objects can then send information back by performing subsequent method invocations on the original caller's unihandle. This typically leads to a pattern of *asynchronous* method calls and callbacks in an M2MI-based application as shown in the examples in Section 4; in other words, an *event-driven* application.

For the same reason, an M2MI method invocation does not give any indication of whether the invocation was successfully communicated to the called objects. If an M2MI-based application needs acknowledgments that a method call in fact reached the called objects, the called objects must do separate method invocations back to the calling object. However, some applications can be designed not to need explicit method acknowledgments at all, achieving fault recovery by other means, as shown in Section 4.

Finally, M2MI invocations are *non-blocking.* A method call on a handle returns immediately to the calling object without waiting for all the target objects to execute their methods. Later, when the method invocations are actually performed, every method in every target object is (potentially) executed concurrently by a separate thread. Therefore, every object exported via M2MI must be designed to be multiple thread safe. Furthermore, like any concurrent application, the overall M2MI-based application must be designed to avoid deadlocks, to work properly with any ordering of the concurrent method calls, and so on.

4 M2MI-Based Systems

This section gives several examples showing how M2MI can be used to design a broad range of ad hoc collaborative systems.

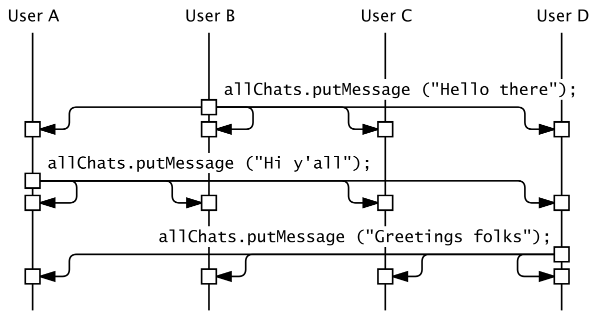

User A
User B
User C
User D
allChats.putMessage ("Hello there");
allChats.putMessage ("Hi y'all");
allChats.putMessage ("Greetings folks");

Figure 8: M2MI invocations for a chat application

4.1 Chat

As a first example of an M2MI-based collaborative system, consider a simple chat session: whenever a user types a line of text, the line is displayed on all the users' devices. Each user's chat application has an object implementing this interface:

```
public interface Chat {
   public void putMessage (String line);
}
```

The application exports the chat object to the M2MI layer. The application also obtains from the M2MI layer an omnihandle for interface `Chat` and stores the omnihandle as `allChats`.

Figure 8 shows a sequence of M2MI invocations that might occur when four instances of this chat application run in four nearby devices. To send a line to everyone in the chat session, the application does a method call on the omnihandle:

```
allChats.putMessage ("Hello there");
```

The chat object's implementation of the `putMessage` method adds the line of text to the chat session log displayed on the user's device. Thus, in response to the above omnihandle invocation, all the exported chat objects display the line of text on all the users' devices.

Note that the M2MI-based chat application does not need to find and connect to a central chat server. Neither does the application need to know which other devices are part of the chat session or connect to them. The user's device simply shows up and starts broadcasting `putMessage` invocations. This shows how M2MI simplifies the development of collaborative systems.

Appendix A gives the actual, working, extremely simple Java code for this M2MI-based chat application.

4.2 Multiple Chat Sessions

Let us add a feature to the chat application: multiple independent simultaneous chat sessions. The user can discover which chat sessions are out there and participate in one of them, or the user can start a new chat session. The user then sees only the messages sent to that chat session, not all the other chat sessions.

To see only the messages for a particular chat session, each user's device has a chat object implementing interface `Chat` as before. Now, however, there is a *multihandle* for interface `Chat` for each separate chat session. To participate in a particular chat session, the application attaches its chat object to the corresponding multihandle. When the user types a line of text, the application invokes `putMessage` on the chat session's multihandle. The chat object processes a `putMessage` invocation exactly as before, by adding the line of text to the chat session log. However, since the invocation is performed on a multihandle instead of an omnihandle, only those chat objects that have been explicitly attached to the multihandle — that is, only those devices participating in the chat session — will respond.

To discover which chat sessions are out there, a new interface is used:

```
public interface ChatDiscovery {
   public void report (Chat session,
      String name);
}
```

The application exports a chat discovery object implementing interface `ChatDiscovery`. Each device with an active chat session periodically invokes `report` on an omnihandle for interface `ChatDiscovery`, passing in the multihandle for the chat session and the name of the chat session. Processing each `report` invocation, the chat discovery object collects the chat sessions in a list and displays them for the user to choose.

If the user decides to participate in an existing chat session, the application obtains the corresponding chat session multihandle from the list and attaches the chat object to the multihandle. If the user decides to start a new chat session, the application creates a new chat session multihandle and attaches the chat object to the multihandle. In either case, the application starts calling `report` periodically.

Figure 9 shows a sequence of M2MI invocations that might occur when four instances of this chat application run in four nearby devices. Users A and C are participating in one chat session named `"AC"`, while users B and D are participating in another named `"BD"`. The corresponding chat session

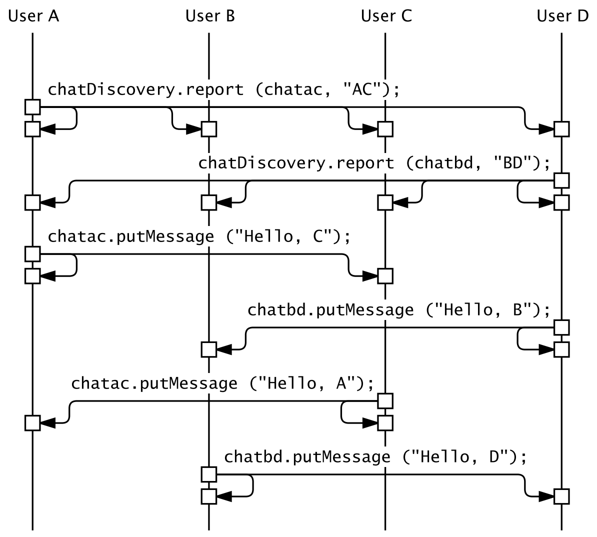

User A
User B
User C
User D
chatDiscovery.report (chatac, "AC");
chatDiscovery.report (chatbd, "BD");
chatac.putMessage ("Hello, C");
chatbd.putMessage ("Hello, B");
chatac.putMessage ("Hello, A");
chatbd.putMessage ("Hello, D");

Figure 9: M2MI invocations for multiple chat sessions

multihandles are named `chatac` and `chatbd`. The omnihandle for interface `ChatDiscovery` is named `chatDiscovery`.

As long as there is at least one device participating in a particular chat session, the periodic `report` invocations for that chat session will continue. When the last participant in the chat session vanishes, the periodic `report` invocations cease. If a certain chat session (multihandle) has not been reported for some amount of time, all the chat discovery objects in all the devices remove that chat session from their lists.

A slight modification of the above scheme will reduce the network traffic. It is not necessary for *every* device participating in a particular chat session to perform `report` invocations for that chat session; only one device need do so. Accordingly, each application starts a timeout for a randomly chosen time interval before doing the next `report` invocation. If someone else calls `report` for the application's chat session before the timeout occurs, the application merely restarts the timeout for another randomly chosen time interval without bothering to call `report` itself. But if the timeout occurs before someone else calls `report` for the application's chat session, the application calls `report` and then restarts the timeout for a randomly chosen time interval.

When a new device arrives in an area where chat sessions are in progress, it may be some time before other devices call `report` and the new device discovers the existing chat sessions. To speed up the discovery process, add a method to the `ChatDiscovery` interface:

```
public interface ChatDiscovery {
   public void request();
   public void report (Chat session,
      String name);
}
```

When the chat application starts up, or when it notices that no one has called `report` for a while, the application calls `request` on an omnihandle for interface `ChatDiscovery`. Processing the invocation, the chat discovery objects in the other devices report their respective chat sessions by calling `report` immediately rather than waiting until the next timeout. However, to avoid a *broadcast storm* [35] where all the devices start calling `report`, saturating the network, only one device in each chat session should respond. Accordingly, every chat discovery object starts a timeout for a small nonzero random amount of time. In each chat session, the first chat discovery object to time out calls `report`. Processing that invocation, the newly arrived device adds the reported chat session to its list as usual, while the other chat discovery objects in that chat session refrain from calling `report` and restart their timeouts for the next report, as usual.

4.3 Chat Log Recovery

Let us add another feature to the chat application. Suppose one or more participants step out of the room, taking their devices with them, so their devices go out of range of the proximal wireless network and no longer receive M2MI invocations. When the participants step back into the room, their devices should automatically fill in the gaps in their chat logs with all the messages they missed while they were out of range, as well as displaying new chat messages. In other words, each device should synchronize its chat log with all the other devices. Let us also assume that each device's chat log only needs to hold the most recent n messages; once the chat log fills up with n messages, older messages don't need to be recovered, and each time a new message is added, the oldest message is deleted.

To add this feature, the chat application needs two things: a way to detect that its chat log does not contain all the messages that another device's chat log contains, and a way to obtain copies of the missing chat messages and put them in their proper places in the chat log.

To detect gaps in the chat log, assign a sequence number to each chat message, and change the `Chat` interface to this:

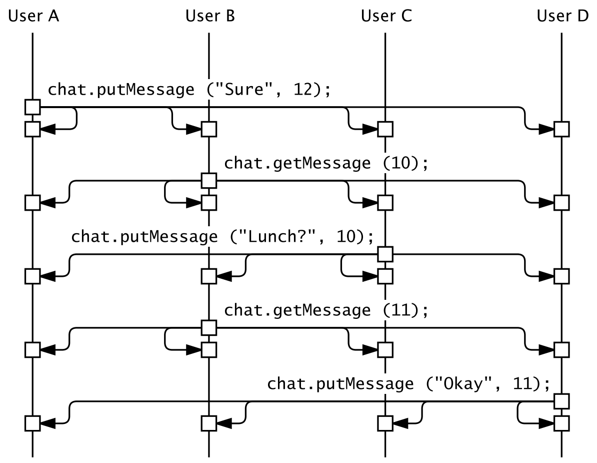

User A
User B
User C
User D
chat.putMessage ("Sure", 12);
chat.getMessage (10);
chat.putMessage ("Lunch?", 10);
chat.getMessage (11);
chat.putMessage ("Okay", 11);

Figure 10: M2MI invocations for chat log recovery

```
public interface Chat {
   public void putMessage (String line,
      long seqnum);
}
```

When a device calls `putMessage`, it passes in the line of text and a sequence number 1 higher than the most recently received chat message. When a device processes a `putMessage` invocation, the device records the new message and its sequence number in the chat log.

If two or more devices call `putMessage` concurrently, then two or more chat messages will end up with the same sequence number. Let us defer dealing with this situation until later and assume for the moment that only one device at a time ever calls `putMessage`.

If the most recently received sequence number is k, and the chat log can hold at most n messages, then the chat log must always contain messages numbered from $\max(1, k-n+1)$ to k. If, after processing a `putMessage` call, a device notices it doesn't have all the chat messages with sequence numbers in that range, the device must synchronize its chat log. To let it do so, the `Chat` interface needs another method:

```
public interface Chat {
   public void putMessage (String line,
      long seqnum);
   public void getMessage (long seqnum);
}
```

Figure 10 shows the sequence of M2MI invocations to synchronize the chat log in device B. (All devices are participating in the same chat session, and all the method invocations are performed on a multihandle for interface `Chat` named `chat`.) Processing a `putMessage` invocation, device B notices it doesn't have all the chat messages implied by the new sequence number. So device B calls `getMessage`, specifying the sequence number of one chat message it needs. To avoid a broadcast storm, only one device should respond. Accordingly, every device's `getMessage` method starts a timeout for a small nonzero random amount of time. The first device to time out calls `putMessage`, passing in the text of the requested chat message and its sequence number. Executing the `putMessage` method, device B records the chat message and its sequence number at the proper place in the chat log, while the other devices cancel their timeouts. Device B continues in this way until all the gaps in its chat log are filled.

A newly arrived device must be able to determine the most recent chat message's sequence number, so the device can pass the correct sequence number in subsequent `putMessage` invocations. If some other device calls `putMessage`, that would supply the needed information. But in case no one is calling `putMessage`, the `Chat` interface needs a third method to let the device request the information explicitly:

```
public interface Chat {
   public void putMessage (String line,
      long seqnum);
   public void getMessage (long seqnum);
   public void getLatestMessage();
}
```

A device calls `getLatestMessage` whenever there have been no M2MI invocations in the chat session for a certain length of time. Responding to the `getLatestMessage` invocation, one of the devices (whichever one times out first) calls `putMessage`, passing in the text of the most recent chat message and its sequence number. The requesting device now knows the most recent sequence number and can also start synchronizing its chat log if necessary.

As will be discussed later, the M2MI invocations may be transported by a network protocol that is not totally reliable, so an invocation may occasionally be lost. To recover from a lost invocation, the device starts a timeout after calling `getMessage` or `getLatestMessage`. If `putMessage` is not called before the timeout, the device retries the invocation; if a certain number of retries all time out, the device gives up.

Now let us return to the issue of multiple chat messages with the same sequence number, resulting from multiple devices calling `putMessage` concurrently. We could impose a protocol to guarantee that every chat message gets a unique sequence number, but that seems hopelessly complicated, especially in an ad hoc network where devices can arrive and depart at any time. Instead, we'll relax the restriction and allow multiple chat messages to have the same sequence number. We'll also allow the devices to display chat messages with the same sequence number in any order, as long as they come after the next lower sequence number and before the next higher sequence number.

Consequently, when responding to a `getMessage` or `getLatestMessage` call, a device may possess more than one chat message corresponding to the requested sequence number. In that case the device calls `putMessage` multiple times, with the same sequence number but different message texts each time.

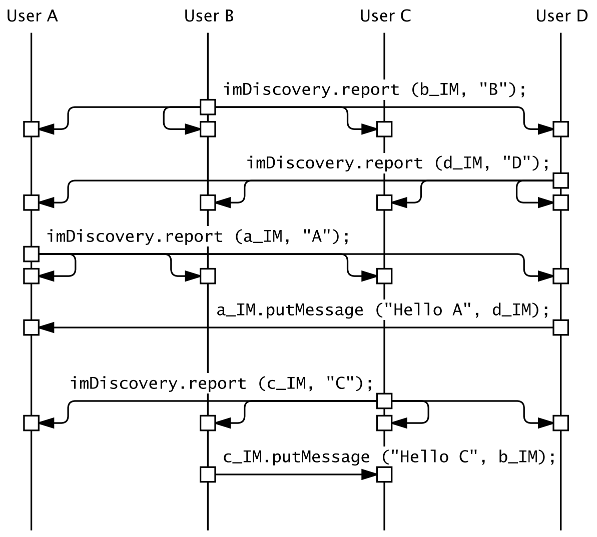
It may also happen that the first device to respond to a `getMessage` or `getLatestMessage` call has a set of chat messages for the requested sequence number that differs from another device's. To handle that case, the other devices monitor the first device's `putMessage` responses. If the other devices detect that they would have responded differently from the first device, the other devices also call `putMessage`.

Finally, it may happen that all the devices' chat logs have the same range of sequence numbers with no gaps, but the chat message texts are different for some sequence number or numbers in different devices. This can happen if the devices separate into multiple groups that are out of wireless range of each other, the chat session continues in each separate group, then the devices come back together again. Since the devices' chat logs all have the same range of sequence numbers, nothing will trigger any device to start a synchronization. To deal with this probably rare case, a device occasionally forces a synchronization by issuing a series of `getMessage` calls for all the sequence numbers in the chat log.

Note that the chat application's log synchronization capability, intended primarily to bring newly arrived devices up to speed, also serves to recover from communication failures. If a device fails to receive a `putMessage` invocation because a network message was lost, on the next `putMessage` invocation the device will detect the missing sequence number and start a synchronization. Even if the network were totally reliable, the chat application would still need the log synchronization capability to deal with newly arrived devices. Therefore, it doesn't make sense to add a lot of code at the network layer to make network communication totally reliable. End-to-end reliability has to be built in at the application level [8].

4.4 Instant Messaging

As another example of an M2MI-based collaborative system, consider a simple instant messaging (IM) system. The IM application needs to discover which



User A
User B
User C
User D
imDiscovery.report (b_IM, "B");
imDiscovery.report (d_IM, "D");
imDiscovery.report (a_IM, "A");
a_IM.putMessage ("Hello A", d_IM);
imDiscovery.report (c_IM, "C");
c_IM.putMessage ("Hello C", b_IM);

Figure 11: M2MI invocations for an IM application

users are out there and send messages to individual users (unlike the chat application which sends messages to all users in a chat session).

The interfaces for the IM application are quite similar to those for the chat application, interface `IMDiscovery` to discover users and interface `IM` to send messages:

```
public interface IMDiscovery {
   public void request();
   public void report (IM user,
      String name);
}
public interface IM {
   public void putMessage (String line,
      IM sender);
}
```

The IM application exports an IM object implementing interface `IM`. But since instant messages go to only one place, the IM object is attached to a *unihandle* (not a multihandle as in the chat application). The IM application also exports an IM discovery object implementing interface `IMDiscovery`.

Figure 11 shows a sequence of M2MI invocations that might occur when four instances of this IM application run in four nearby devices. Each application announces its presence by calling `report` on an omnihandle for interface `IMDiscovery`, passing in the unihandle to its own IM object and its own user's name. For example, user A's application calls:

```
imDiscovery.report (a_IM, "A");
```

Executing the `report` method, each IM discovery object stores the unihandle and the user name in an internal list for later use.

To send an instant message to a particular user, the application looks up the corresponding `IM` unihandle in the user list and calls the `putMessage` method on the unihandle, passing in the message text and the unihandle to its own IM object (so the recipient knows who sent the message). For example, to send an instant message to user A, user D's application calls:

```
a_IM.putMessage ("Hello A", d_IM);
```

The `putMessage` method displays the message and the sender on the destination device's display. Since the invocation is performed on a unihandle, not a multihandle or omnihandle, only the destined user's IM object executes the `putMessage` method and displays the message; the message does not appear on the other devices' displays.

To show that the user is still present, each instance of the IM application broadcasts a `report` invocation periodically on an omnihandle for interface `IMDiscovery`. If the time since the last `report` invocation for a certain user (unihandle) exceeds a threshold, the other IM applications conclude the user has gone away and remove the user from their user lists. To quickly discover which users are present, a device invokes `request` on an omnihandle for interface `IMDiscovery`, and all the IM discovery objects respond by calling `report` immediately.

4.5 Service Discovery — Printing

As an example of an M2MI-based system involving standalone devices providing services, consider printing. To print a document from a mobile device, the user must discover the nearby printers and print the document on one selected printer. Printer discovery is a two-step process: the user broadcasts a printer discovery request via an omnihandle invocation, then each printer sends its own unihandle back to the user via a unihandle invocation on the user. To print the document, the user does an invocation on the selected printer's unihandle.

Specifically, each printer has a print service object that implements this interface:

```
public interface PrintService {
   public void print (Document doc);
}
```

The printer exports its print service object to the M2MI layer and obtains a unihandle attached to the object. The printer is now prepared to process document printing requests.

To discover printers, there are two print discovery interfaces:

```
public interface PrintDiscovery {
   public void request
      (PrintClient client);
}
public interface PrintClient {
   public void report
      (PrintService printer,
       String name);
}
```

In the chat and IM applications, the participating devices all played the same roles, both making discovery requests and generating discovery reports. In the printing application, though, the participating devices do not play the same roles. Some devices are clients which make discovery requests but do not generate discovery reports; other devices are printers which generate discovery reports but do not make discovery requests. Accordingly, in the printing system there is a separate discovery interface for each role.

The client printing application exports a print client object implementing interface `PrintClient` to the M2MI layer and obtains a unihandle attached to the object. The application also obtains from the M2MI layer an omnihandle for interface `PrintDiscovery`. The application is now prepared to make print discovery requests and process print discovery reports.

Each printer exports a print discovery object implementing interface `PrintDiscovery` to the M2MI layer. The printer is now prepared to process print discovery requests and generate print discovery reports.

Figure 12 shows the sequence of M2MI invocations that occur when the document printing application goes to print a document with three printers nearby. The application first calls

```
printDiscovery.request (theClient);
```

on an omnihandle for interface `PrintDiscovery`, passing in the unihandle to its own print client object. Since it is invoked on an omnihandle, this call goes to all the printers. The application now waits for print discovery reports.

Each printer's `request` method calls

```
theClient.report (thePrinter,
   "Printer Name");
```

The method is invoked on the print client unihandle passed in as an argument. The method call arguments are the unihandle to the printer's print service object and the name of the printer. Since it is invoked on a unihandle, this call goes just to the requesting client printing application, not to any other print clients that may be present. After executing all the `report` invocations, the printing application knows the name of each available printer and has a

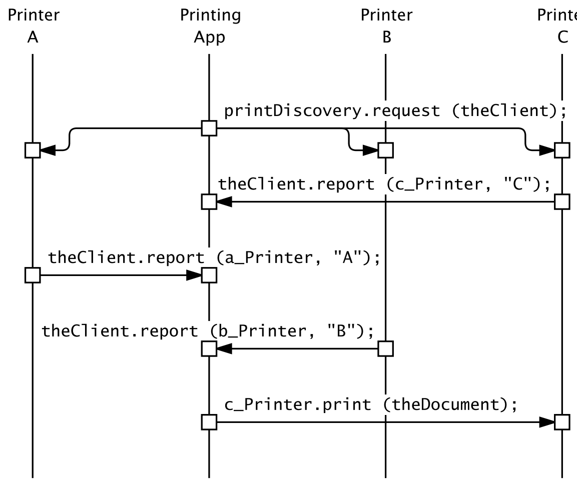

Printer A
Printing App
Printer B
Printer C
printDiscovery.request (theClient);
theClient.report (c_Printer, "C");
theClient.report (a_Printer, "A");
theClient.report (b_Printer, "B");
c_Printer.print (theDocument);

Figure 12: M2MI invocations for a print service

unihandle for submitting jobs to each printer.

Finally, after asking the user to select one of the printers, the application calls

```
c_Printer.print (theDocument);
```

where `c_Printer` is the selected printer's unihandle as previously passed to the `report` method. Since it is invoked on a unihandle, this call goes just to the selected printer, not the other printers. The printer proceeds to print the document passed to the `print` method.

Clearly, this invocation pattern of broadcast discovery request – discovery responses – service usage can apply to any service, not just printing. It is even possible to define a *generic* service discovery interface that can be used to find objects that implement *any* interface, the desired interface being specified as a parameter of the discovery method invocation.

4.6 Advanced Printing

When printing a document, the user may need the printer to have certain features — such as the ability to print multiple copies of a document, or the ability to staple the pages, or having a certain size of paper loaded. Alternatively, the user may need the printer to be in a certain state — such as not jammed, or not too many jobs backed up in the print queue. In such cases, the user wants to discover only the printers that fulfill the user's criteria, not all the printers. Furthermore, when actually printing the document, the user wants to specify the number of copies, stapling, paper size, and other characteristics of the print job in addition to the document itself.

To accomplish this, add some methods to the `PrintDiscovery` interface and to the `PrintService` interface:

```
public interface PrintDiscovery {
   public void request
      (PrintClient client);
   public void request
      (PrintClient client,
       Attribute attr);
   public void request
      (PrintClient client,
       AttributeSet attrs);
}
public interface PrintService {
   public void print (Document doc);
   public void print (Document doc,
      Attribute attr);
   public void print (Document doc,
      AttributeSet attrs);
}
```

The various printer characteristics — copies, stapling, paper, printer status, print queue status, and so on — are all represented as *attributes.*[1] To discover printers that have, say, ISO A4 paper loaded, the printing application invokes the second `request` method instead of the first, passing in the desired attribute:

```
printDiscovery.request (theClient,
   MediaSize.ISO.A4);
```

While all the printers still execute this method in response to the omnihandle invocation, only those printers that match the attribute — namely, those that have ISO A4 paper loaded — will call back to the client. Likewise, to discover printers that support multiple attributes, the printing application invokes the third `request` method, passing in a set of the desired attributes; only those printers that match all the attributes will respond. Consequently, the client becomes aware of just those printers that match the user's requirements.

To specify job characteristics for the actual print job, the printing application invokes the second or third `print` method instead of the first, passing in the desired attribute or set of attributes:

```
c_Printer.print (theDocument,
   MediaSize.ISO.A4);
```

This example shows that, by defining the appropriate interfaces, service discovery can be tailored specifically for the service.

[1]The Internet Printing Protocol (IPP) [19] defines a rich set of printing attributes. For examples of Java APIs that encapsulate the IPP printing attributes, see the Jini Print Service API [22] and the Java Print API [47].

4.7 File Sharing

As a final example of an M2MI-based collaborative system, imagine a file sharing application. Each user's device has a number of files which the user is willing to share. When the file sharing application runs among a group of proximal devices, the user sees all available shared files — that is, the union of the sets of shared files in all the devices. If a certain file exists on more than one device, that file shows up only once in the application. As devices enter and leave the proximal group, the set of shared files visible on each device grows and shrinks. The user can get information about any shared file, such as its name, its type, its size, a textual annotation, a thumbnail view, and so on. The user can also browse the contents of any shared file — read a text file, view an image file, play a sound file.

When a device leaves the proximal group, from that device's point of view all the shared files disappear except for those stored on the device itself. However, before leaving, the user can tell the file sharing application to *keep* a certain file or files. The application stores a copy of those files on the user's device (which does not change the set of shared files as viewed by all the devices). Now, however, the kept files do not disappear when the device leaves the proximal group.

An ad hoc collaborative file sharing application can be used in many ways. Spectators in public settings like athletic competitions, sporting events, amusement parks, and scenic places can share the digital photos they're all taking. A group of friends can listen to one another's music files, or swap copies of music files.[2] Businesspeople in a meeting can share reports, presentations, contact information, and so on.

To detect whether the same file exists on multiple devices without having to transfer the entire files over the network, the file sharing application uses a *one-way hash of the file's contents* (such as an MD5 hash [43] or SHA-1 hash [34]) to uniquely identify a file. Meta-information about a file, such as its name or type, is not part of the file's contents. Thus, two files with different names but the same contents will have the same IDs (hashes) and will be considered the same file.

Each file sharing application exports an object implementing this interface:

```
public interface FileShare {
   public void available (Hash[] ids);
   public void requestFile (Hash id);
```

[2]Always provided, of course, that the files are legally allowed to be copied.

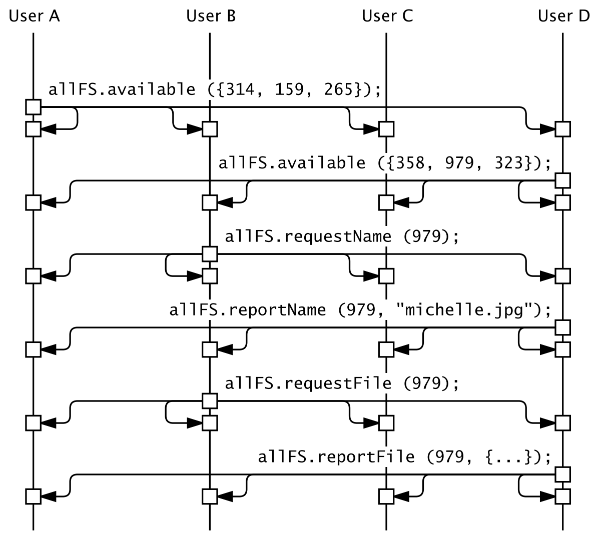

User A
User B
User C
User D
allFS.available ({314, 159, 265});
allFS.available ({358, 979, 323});
allFS.requestName (979);
allFS.reportName (979, "michelle.jpg");
allFS.requestFile (979);
allFS.reportFile (979, {...});

Figure 13: M2MI invocations for a file sharing application

```
    public void reportFile (Hash id,
       byte[] contents);
    public void requestName (Hash id);
    public void reportName (Hash id,
       String name);
}
```

Figure 13 shows a sequence of M2MI invocations that might occur when four instances of this file sharing application run in four nearby devices. Every device periodically performs an omnihandle invocation of the `available` method, passing in an array of the IDs of the files it has to share. (The array of hashes takes much less time to transmit than the files themselves.) Processing the `available` method, each device adds the specified IDs to its list of available IDs, except for those already in the list. Each device also starts or re-starts a timeout for each of the specified IDs. If a device leaves the proximal group, the device stops doing `available` invocations, the IDs for that device's files time out (unless some other device is reporting they're still available), and each remaining device removes those IDs from its list of available IDs.

To find out further information about a particular file, such as its name, the device performs an omnihandle invocation of the `requestName` method, passing in the desired file's ID. To prevent a broadcast storm if multiple devices possess that file, the `requestName` method in each device that has the file starts a small nonzero random timeout. The first device to time out performs an omnihandle invocation of the `reportName` method, passing in the requested file's ID and name; the other devices if any cancel their timeouts. The requesting device now has the file name. Clearly, any other piece of meta-information can be provided by adding the appropriate `request` and `report` methods to the `FileShare` interface.

In the same way, to get the actual contents of a particular file, the device calls `requestFile` on the omnihandle. One of the devices possessing the file then calls `reportFile` on the omnihandle, passing in the file's contents. Executing the `reportFile` method, the requesting device can display the file, store it locally on the device (to keep the file), and so on.

Besides reducing the time needed to transmit unique file identifiers around the network, using one-way hashes to identify files provides a measure of security. An intruder could try to disrupt the file sharing application by sending some file other than the requested file in a `reportFile` invocation. However, if the reported ID (hash) does not match the actual hash of the reported contents, the recipient knows the contents are not correct and can discard them. While the intruder's bogus `reportFile` invocation does consume network and processing resources, it does not cause the application to behave incorrectly.

Two things need improving in the file sharing application presented so far. First, because all the method invocations were performed on omnihandles, every device received every file's meta-information and contents when requested by any device. In the case of meta-information this behavior is desirable, since every device will likely need to display the meta-information for every shared file. In the case of file contents this behavior is less desirable. When a device which did not request the file's contents executes the `reportFile` method in response to an omnihandle invocation, the device could nonetheless capture the file and save it for possible later use, or the device could simply do nothing. However, if not every device is going to need the contents of every shared file, it would be better not to send the file's contents to every device.

The second problem is that in the `reportFile` method defined above, the entire contents of the file was passed all at once. However, especially for a large file, the receiving device may not have enough buffer space to hold the entire file all at once. Also, if a communication failure occurs while the `reportFile` invocation is traveling through the network, the entire contents will have to be sent again, which wastes bandwidth.

To solve both problems, define two additional interfaces, one for the device sending a file and one for the device receiving it:

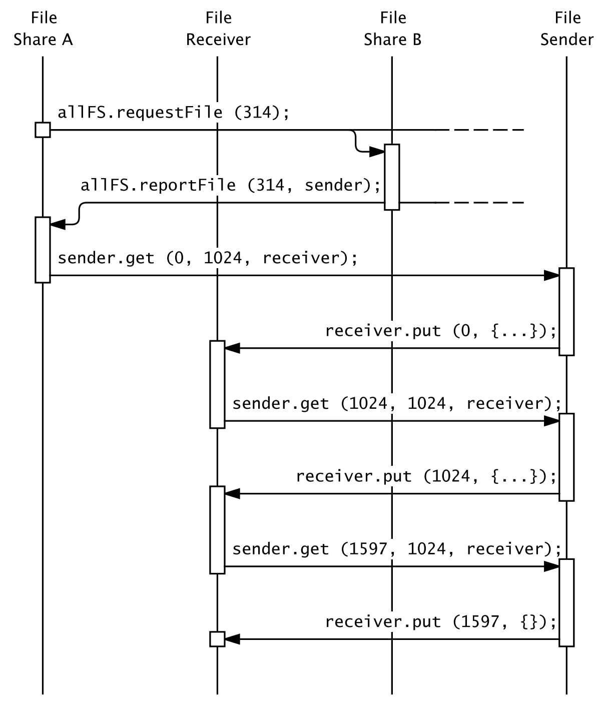

File Share A
File Receiver
File Share B
File Sender
allFS.requestFile (314);
allFS.reportFile (314, sender);
sender.get (0, 1024, receiver);
receiver.put (0, {...});
sender.get (1024, 1024, receiver);
receiver.put (1024, {...});
sender.get (1597, 1024, receiver);
receiver.put (1597, {});

Figure 14: M2MI invocations for transferring a file

```
public interface FileSender {
   public void get (long offset,
      int count, FileReceiver receiver);
}
public interface FileReceiver {
   public void put (long offset,
      byte[] contents);
}
```

Also, change the interface of the `reportFile` method:

```
public interface FileShare {
   public void available (Hash[] ids);
   public void requestFile (Hash id);
   public void reportFile (Hash id,
      FileSender sender);
   public void requestName (Hash id);
   public void reportName (Hash id,
      String name);
}
```

Figure 14 shows the sequence of M2MI invocations that occurs when device *A* gets a file from device *B* using the revised interfaces. Device *A* starts by calling `requestFile` on the omnihandle, passing in the desired file ID. The device which possesses that file, *B*, executing `requestFile`, creates a `FileSender` object for that file, exports the object to the M2MI layer, and obtains a unihandle. Then *B* calls `reportFile` on the omnihandle, passing in the file ID and the file sender object's unihandle. *A*, executing `reportFile`, saves the file sender object's unihandle. Then *A* creates a `FileReceiver` object for the file, exports the object to the M2MI layer, and obtains a unihandle. Finally, *A* calls `get` on the file sender object's unihandle to get the first chunk of the file. The arguments to `get` are the offset of the first byte in the chunk, the number of bytes in the chunk (a size that *A* can conveniently handle), and the file receiver object's unihandle. *B*, executing `get`, calls `put` on the file receiver object's unihandle, passing in the starting offset of the chunk and the contents of the chunk. *A*, executing `put`, stores the chunk and calls `get` to obtain the next chunk. This sequence of alternating `get` and `put` calls continues until the entire file has been transferred. *B* signals the end of the file by calling `put` with a zero-length chunk.

To recover from communication failures, *A* starts a timeout after calling `get`. If the chunk does not arrive in a `put` call before the timeout, *A* retries the `get`; if a number of retries all fail, *A* gives up.

Once the entire file has been transferred (or an unrecoverable failure has happened), *A* can destroy the file receiver object. *B* can destroy the file sender object once a certain amount of time has elapsed with no `get` calls.

4.8 Summary

The examples in this section have shown how objects implementing simple interfaces, coupled with M2MI's ability to invoke methods on many objects at once, can be used to build different kinds of powerful and interesting ad hoc collaborative systems. None of the systems required central servers; none of the systems required knowledge of individual device addresses. All of the systems allowed new devices to join the collaborative group simply by showing up and starting to broadcast M2MI invocations, without needing to perform explicit discovery or group joining protocols. M2MI is thus well suited to ad hoc networks of small mobile devices.

5 M2MI Architecture

Our initial work with M2MI has focused on networked collaborative systems. In this environment of ad hoc networks of proximal mobile wireless devices, M2MI is layered on top of a new network protocol, M2MP. We have implemented initial versions of M2MP and M2MI in Java.

Figure 15 shows the overall architecture of M2MI. When the calling object invokes a target method on a handle, the invocation may have to be performed

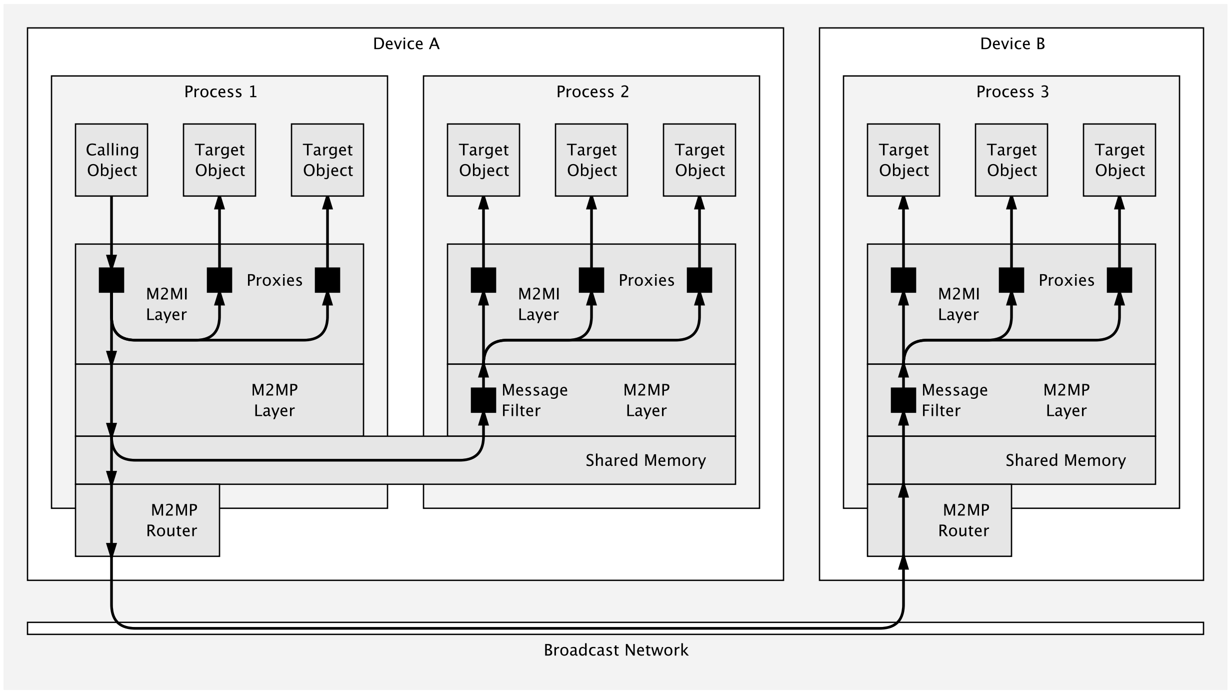


Figure 15: M2MI Architecture

by target objects in three places: in the same process as the calling object, in different processes in the same device, and in different devices. The invocation travels along different paths to the three destinations.

Each process that employs M2MI has a singleton instance of the M2MI layer, and the M2MI layer has an instance of the M2MP layer. When the calling object invokes a target method on a handle, the handle forwards the invocation to the M2MI layer in its own process. The M2MI layer in turn forwards the invocation to the appropriate objects that have been exported to the M2MI layer in that process, if any. No messages are sent out of the process to reach these objects.

To reach target objects in other processes, the M2MI layer transmits the invocation in the form of a message (byte stream) to the M2MP layer. All the M2MP layers in the same device share a region of memory. The transmitting M2MP layer deposits the invocation message into the shared memory. The other M2MP layers each obtain a copy of the invocation message from the shared memory and pass the message up to their respective M2MP layers. No messages are sent out of the device onto an external network to reach these objects.

Before reaching the M2MI layer, however, the invocation message must pass through a *message filter* in the M2MP layer. Only invocation messages destined for target objects exported in the M2MI layer in that process will pass through the message filter; messages destined for target objects in other processes will be discarded.

To reach target objects in other devices, an *M2MP router* in each device listens to the M2MP shared memory and transmits each outgoing message on the external broadcast network. The message is broadcast to all the devices in the proximal network. The M2MP router also listens to the external network and injects each incoming message into the M2MP shared memory, whence the message is processed in the same way as messages originating in the same device.

Thus an M2MI invocation is broadcast through the M2MI layer to all target objects in the same process; is broadcast through the shared memory to all target objects in other processes in the same device; and is broadcast through the external network to all target objects in other devices. The M2MP message filters weed out irrelevant messages, letting the M2MI layers devote processing resources only to the relevant messages.

In a device that does not have multiple processes, such as a small handheld device, the M2MI architecture is simpler. The shared memory and M2MP router are omitted. The M2MP layer sends outgoing messages directly to the external network and receives incoming messages directly from the external network.

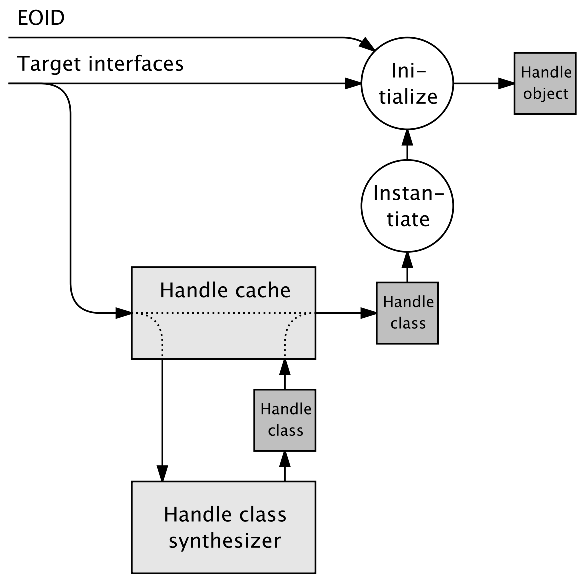


Figure 16: Creating a handle object

6 M2MI Design

This section describes the design of the M2MI layer at a high level. The M2MI API is described in the documentation that accompanies the M2MI software [25].

6.1 Handles

An M2MI handle object encapsulates two pieces of information: a list of the fully-qualified names of the handle's target interfaces (one or more), and a 128-bit *exported object identifier* (EOID). For an omnihandle, the EOID is a wildcard (zero). For a unihandle, the EOID is a globally unique nonzero value that designates a single exported object. For a multihandle, the EOID is a globally unique nonzero value that designates a particular set of exported objects.

A handle object's class implements all of the handle's target interfaces, providing an implementation for every method in every target interface and all superinterfaces thereof. A handle object can thus be cast to any of its target interfaces, and any method in any target interface can be invoked on a handle object.

When a handle object needs to be created (Figure 16), the M2MI layer first *synthesizes* the handle class. The M2MI layer uses Java reflection to identify all the target methods, creates a byte array containing a binary class file with implementations for all the target methods, loads the class file byte array into a special class loader, and gets back the handle class. To do this, the M2MI layer employs the RIT Classfile Library [27], a general purpose library for synthesizing Java class files. The M2MI layer then stores the handle class in a cache. The next time a handle is needed for the same target interfaces, the M2MI layer gets the handle class from the cache instead of synthesizing it again. Having obtained the handle class, the M2MI layer creates an instance of it and stores the proper target interface names and EOID in the newly created handle object.

An alternative to synthesizing handle classes would be to implement handles using Java reflection's dynamic proxies (class `java.lang.reflect.Proxy`). Measurements on several M2MI-based applications showed, however, that the applications ran 5 to 30 percent faster when the M2MI layer was implemented with synthesized classes than when the M2MI layer was implemented with dynamic proxies.

6.2 Exporting Objects

An object can be exported to the M2MI layer by calling `M2MI.export`, giving the object and the target interface or interfaces. In response, the M2MI layer adds the object to the *interface export map*, which maps the fully-qualified name of a target interface to a set of objects that have been exported as that target interface. For each target interface and each superinterface thereof, the object is added to the corresponding set in the interface export map. This lets the object be invoked by an omnihandle as described later.

An object can also be exported to the M2MI layer by calling `M2MI.getUnihandle`, giving the object and the target interface or interfaces. In response, the M2MI layer adds the object to the interface export map as before. The M2MI layer also adds the object to the *EOID export map*, which maps an EOID to a set of exported objects associated with that EOID. The M2MI layer generates a new EOID, adds the (EOID, object) mapping to the EOID export map, and returns a unihandle for the target interfaces initialized with that EOID. This lets the object be invoked by that unihandle as described later, as well as by an omnihandle. The M2MI layer will ensure that that EOID only ever maps to one object.

Finally, an object can be exported to the M2MI layer by first calling `M2MI.getMultihandle` to get a multihandle for a certain target interface or interfaces, then calling `attach` on the multihandle giving the object. To create a multihandle, the M2MI layer generates a new EOID and returns a multihandle for the target interfaces initialized with that EOID. When an object is attached to the multihandle, the

M2MI layer adds the object to the set of objects associated with the multihandle's EOID in the EOID export map, and the M2MI layer adds the object to the interface export map as before. This lets the object be invoked by that multihandle as described later, as well as by an omnihandle.

6.3 Performing an M2MI Invocation

An M2MI invocation starts when the calling object invokes a target method in a target interface on a handle. The target method uses Java object serialization to serialize the method's arguments, if any, into a byte array. The target method then passes the following information to the M2MI layer: the handle's EOID (initialized when the handle was created), the fully-qualified name of the target interface, the target method's name, the target method's descriptor, and the serialized arguments (Figure 17).

The M2MI layer needs to set up *method invoker* objects that will ultimately perform the invocations on the target objects. A method invoker is a `Runnable` object whose `run` method is customized to invoke a certain method in a certain interface. The M2MI layer *synthesizes* the appropriate method invoker class for the given target interface name, target method name, and target method descriptor. The M2MI layer saves the method invoker class in a cache to be retrieved the next time a method invoker is needed for the same target method and interface.

The M2MI layer next needs to find the target objects for the invocation that have been exported in the M2MI layer's process. For an omnihandle invocation, the M2MI layer uses the interface export map to map the target interface name to the set of target objects. For a unihandle or multihandle invocation, the M2MI layer uses the EOID export map to map the EOID to the set of target objects.

For each target object, the M2MI layer creates an instance of the method invoker class. The method invoker object is initialized with a reference to the target object and a reference to the byte array containing the serialized arguments. The method invoker object is then placed in a work queue for further processing on a separate thread.

Finally, the M2MI layer uses the M2MP layer to send an outgoing invocation message as detailed in Section 7.6. The M2MP layer is responsible for broadcasting the invocation message to other processes and/or devices. However, if the invocation was performed on a unihandle for an object exported in the M2MI layer's process, the M2MI layer does not send an outgoing invocation message (because there are no other target objects that need to be invoked).

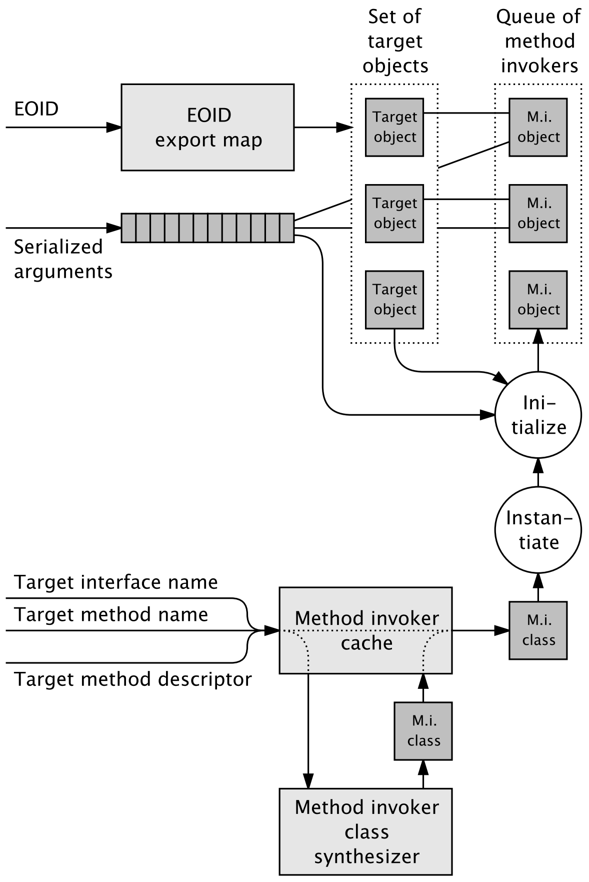


Figure 17: Performing an M2MI invocation, part 1

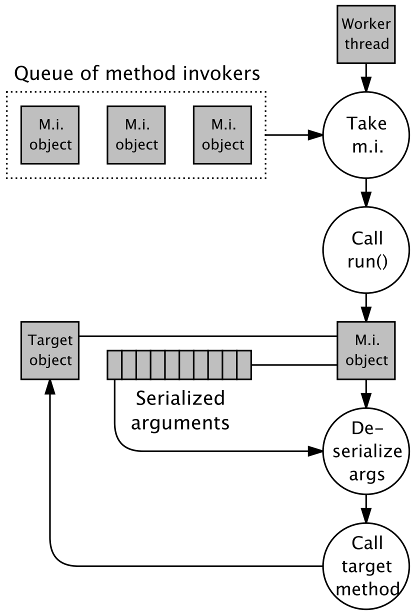


Figure 18: Performing an M2MI invocation, part 2

At this point the original call on the handle returns to the calling object.

The M2MI layer has a pool of one or more worker threads to process the work queue (Figure 18). (The number of worker threads needed depends on the application and is established when the M2MI layer is initialized.) Concurrently, a worker thread takes a method invoker object from the head of the work queue and calls the method invoker's `run` method. The `run` method deserializes the method arguments from the byte array with which the method invoker was initialized. The `run` method then invokes the target method on the target object with which the method invoker was initialized, passing in the deserialized arguments. Once the target method returns, the worker thread goes on to the next method invoker in the queue, blocking if necessary until one is added to the queue.

Since each method invoker separately deserializes the arguments from the byte array of serialized arguments, each target method gets its own copies of the arguments separate from the original calling object's arguments and separate from the other target objects' arguments. This enforces M2MI's argument pass-by-copy semantics.

6.4 Serialization of Handles

A handle object can be passed as an argument in an M2MI invocation just like any other object. However, handle objects must be treated specially during serialization and deserialization to ensure they work properly when reconstituted at the destination, which might in be a different process or device from the calling object. To do this, M2MI uses Java object serialization's *object replacement* capability [46].

Each handle class includes a `writeReplace` method. When a handle is serialized, the serialization system notices the `writeReplace` method and calls it. The `writeReplace` method returns a *handle transport object* initialized with the handle's EOID and target interface list. The serialization system then serializes this handle transport object rather than the original handle.

The handle transport class includes a `readResolve` method. When the handle transport object is deserialized at the destination, the serialization system notices the `readResolve` method and calls it. The `readResolve` method tells the destination's M2MI layer to create a handle for the target interfaces and EOID stored in the handle transport object. The M2MI layer creates the handle as usual, synthesizing the handle class if necessary. The `readResolve` method returns the handle, and the serialization system returns the handle as the result of the deserialization (instead of the handle transport object). Thus, the destination ends up with a handle for the same target interfaces and EOID as the original handle.

7 M2MP Design

This section describes the design of the M2MP layer at a high level. The M2MP API is described in the documentation that accompanies the M2MI software [25]. After describing the M2MP design, this section also describes how the M2MI layer uses the M2MP layer.

7.1 Assumptions

Intended particularly for the wireless proximal ad hoc networking environment, M2MP's design is based on these assumptions:

- *There are no device addresses.* Consequently, devices can enter and leave the network in an ad hoc fashion without having to maintain any routing tables.
- *Messages are broadcast to all devices.* Since wireless radio transmissions are inherently broadcast within a certain proximal area, at the radio level it's just as easy to deliver a message to all devices as to one device.
- *A message's relevancy is determined by its contents.* A device decides which incoming messages to process by examining the initial bytes of each message.
- *Message delivery is mostly reliable.* Most of the time, a message broadcast by one device is received by all the other devices. However, on rare occasions a message broadcast by one device is not received by some or all of the other devices.

7.2 Outgoing Messages

When an application on one device sends an M2MP message, the application writes a stream of bytes with the message's contents to the M2MP layer. The M2MP layer breaks the byte stream into a sequence of fragments, wraps each fragment in a packet, and broadcasts each packet. An M2MP packet consists of these fields:

- Message ID (4 bytes)
- Fragment number and last packet flag (4 bytes)

- Message fragment (N bytes)
- Checksum (2 bytes)

The maximum length of an M2MP packet is 508 bytes. This number is chosen so an M2MP packet can be transmitted as a single datagram without fragmentation by various underlying transport protocols. Thus, the maximum length of each message fragment is 498 bytes.

To let the receiving devices reassemble packets sent simultaneously by many transmitting devices into the proper messages, every packet of an M2MP message carries the same value in the message ID field. Each device generates message IDs for successive messages by stepping through a random permutation of the set of 32-bit integers. Each device seeds its permutation generator with unique information including the device's system clock value and the device's physical layer address (such as an Ethernet MAC address or a Bluetooth device address). Thus, each device generates a different permutation of the integers, and there is a negligible probability that two devices will generate the same message ID at the same time. In this way, packets from different messages can be distinguished without the devices having to coordinate with each other.

The fragments of a message are numbered starting with 0, and the fragment number field identifies which fragment the packet contains. The last packet flag is 0 in all packets except the last, where it is 1.

The message fragment field holds the message fragment itself. Each message fragment except possibly the last is 498 bytes long. The length of the message fragment, N, is not carried within the packet. Instead, the overall packet length is obtained from the next lower protocol layer used to transport the packet, and this determines N.

Finally, the checksum field contains a simple 16-bit ones complement sum of the rest of the packet and is used to detect alteration of the packet during transit. (This checksum is unable to detect certain kinds of attack and must be strengthened as discussed in Section 9.1.)

7.3 Incoming Messages

To receive incoming messages, an application must register one or more *message filters* with the M2MP layer. Each message filter has a *message prefix,* a fixed byte string. If an incoming message's initial bytes match the message prefix of a registered message filter, the M2MP layer passes the message up to the application that registered the message filter. Otherwise, the M2MP layer discards the message, and the application never sees it. An application that uses M2MP, such as M2MI, designs the contents of its M2MP messages to take advantage of M2MP's message filtering capability and weed out irrelevant messages before they ever reach the application.

The M2MP layer processes each incoming packet as follows. If the checksum is not correct, the packet is discarded as corrupt. If it is the first packet of a message (fragment number is 0), the M2MP layer compares the message fragment to all the registered message filters' message prefixes using an efficient trie search. If there is no match, the packet is discarded as irrelevant. But if there is a match, the M2MP layer creates a new incoming message associated with the packet's message ID and forwards the message to the application that registered the matching message filter. The application reads a stream of bytes containing the message's contents, beginning with the message fragment in the first packet. If there are further packets in the message (last packet flag is 0), the M2MP layer starts a timeout to wait for the next packet.

If an incoming packet is not the first packet of a message (fragment number is greater than 0), the M2MP layer looks for an in-progress message associated with the packet's message ID. If there is none, the packet is discarded as irrelevant. If there is a message in progress, but the packet's fragment number is not the next expected fragment number, the packet is discarded as out of sequence. Otherwise, the M2MP layer cancels the timeout and feeds the packet's message fragment to the application reading the message. If there are further packets in the message, the M2MP layer restarts the timeout to wait for the next packet.

If a failure occurs in the middle of a message, such as a lost packet or a corrupted packet, the M2MP layer will time out waiting for the packet with the expected next fragment number to arrive. If the timeout occurs, the M2MP layer abandons the message and signals an exception to the application reading the message. The M2MP layer neither acknowledges nor retransmits packets.

Retransmitting lost packets is unnecessary, and abandoning the message is acceptable, because we assume the proximal network is mostly reliable. Recovery from an occasional message loss can be done at the application level. Indeed, the messaging layer should not be expected to provide end-to-end delivery or ordering guarantees [8]. This considerably simplifies M2MP.

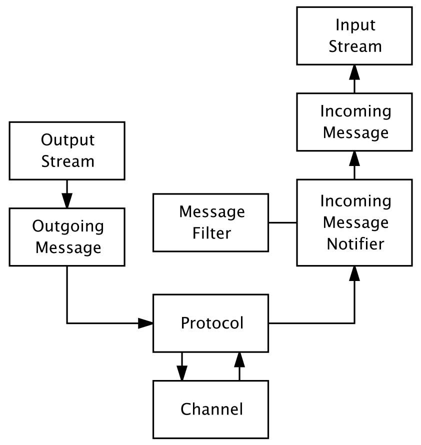


Figure 19: Objects in the M2MP API

7.4 M2MP API

Figure 19 shows the principal objects in the M2MP API and the patterns of data flow among them. The protocol object forms the core of M2MP. The channel object interfaces the protocol core to the underlying layer. By plugging a different channel object into the protocol core, M2MP can be used with different underlying protocols. The remaining objects interface the rest of the application to the protocol core.

To send an M2MP message, the application creates an outgoing message object, obtains an output stream from the outgoing message, and writes the message's contents to the output stream.

To receive M2MP messages, the application creates an incoming message notifier object. The application registers one or more message filter objects with the incoming message notifier. The application reads incoming message objects from the incoming message notifier, which returns only those messages that match one of the message filters. For each incoming message, the application obtains an input stream and reads the message's contents from the input stream.

7.5 Underlying Layers

The layer shown in Figure 15 underneath the M2MP layer is presently implemented by a channel object that uses the Internet protocol stack in lieu of shared memory (Figure 20). The channel wraps each outgoing M2MP packet in a UDP datagram and sends the datagram to a designated well-known port on the local host IP address, 127.0.0.1. Concurrently, the channel reads UDP datagrams containing incoming M2MP packets from its own separate port on the local host IP address. A separate M2MP router process receives datagrams from the well-known port

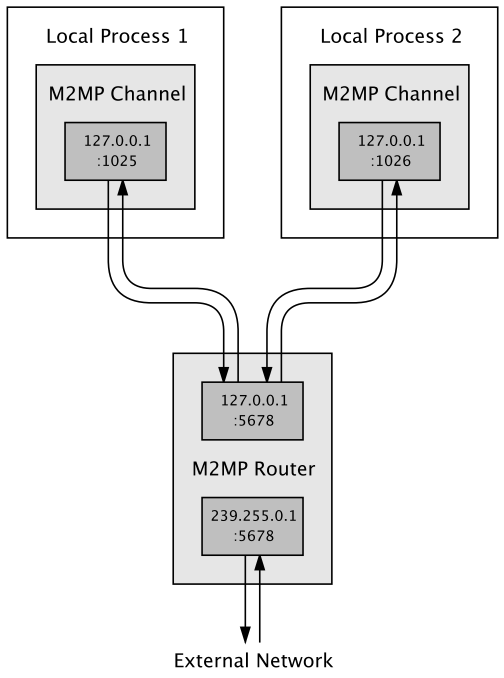


Figure 20: M2MP channels and M2MP router

and sends copies of them to all the channels' ports. Although it is a roundabout way of achieving interprocess broadcast, this scheme can be implemented in pure Java without needing nonportable native code libraries or operating system kernel modifications.

Additionally, the M2MP router process sends a copy of each datagram from a local process to an external network address, and concurrently receives incoming datagrams from an external network address and copies them to the local processes. The external network address can be a unicast IP address, in which case M2MP messages are tunneled between just two devices. Alternatively, the external network address can be a multicast IP address, in which case M2MP messages are broadcast to all devices that have joined the multicast group.

Ideally, M2MP would be supported directly by the operating system with its own protocol stack, including a true shared memory layer, and would not have to incur the additional overhead of the Internet protocol stack. Adding M2MP support to the operating system kernel is an area of future work.

7.6 M2MI's Use of M2MP

Having described the M2MP layer's design, we can now describe how the M2MI layer uses the M2MP layer.

When a calling object calls a target method on a handle, the M2MI layer sends an outgoing invocation as an M2MP message, and the M2MP layer broadcasts this message to all processes and devices. The invocation message contains these items:

- Magic number, `"M2MI"` in ASCII (4 bytes)
- Hash code of the key used to find the target objects in the interface export map or EOID export map (4 bytes)
- Target interface name (UTF-8 string)
- Target method name (UTF-8 string)
- Target method descriptor (UTF-8 string)
- Length of the serialized arguments (4 bytes)
- The serialized arguments themselves, if any

In an M2MI invocation message, the message prefix used for message filtering consists of the first 8 bytes: the magic number and the key's hash code. Whenever a target object is exported — that is, whenever a target object is associated with a certain key in either the interface export map or the EOID export map — the M2MI layer registers a message filter with the corresponding message prefix. Likewise, whenever a target object is unexported, the M2MI layer deregisters the message filter with the corresponding message prefix. The M2MP layer's trie data structure allows the message prefixes to be stored and searched efficiently even if many objects are exported.

When an incoming M2MP message containing an M2MI invocation arrives at the M2MP layer, the M2MP layer compares the message's initial bytes to the registered message prefixes. If the magic number doesn't match, the message was not generated by an application using M2MI, and the message can be discarded. If the key's hash code doesn't match, then the invocation is not destined for any target object exported in this process, and the message can be discarded.

If both the magic number and the key's hash code match, the M2MP layer passes the invocation message on up to the M2MI layer. The M2MI layer skips over the message prefix (which is there only for efficient message filtering in the M2MP layer) and extracts the target interface name, target method name, target method descriptor, and serialized arguments. The M2MI layer then proceeds to process the invocation in exactly the same way as an invocation originating within its own process.

8 Related Work

M2MI touches on several areas of related work, including ad hoc networking, remote method invocation, distributed systems architecture, and collaborative middleware.

8.1 Ad Hoc Networking

A considerable amount of work has been done on ad hoc networking. This work has concentrated on how to make networking based on host addresses (such as IP addresses) work when hosts move around and do not stay attached to a fixed network segment. Mobile IP [21], for example, is a scheme where a host can move to a different location, obtain a temporary IP address there, and cause traffic sent to the host's permanent address to be forwarded to its temporary address. Many ad hoc routing algorithms have been developed to route messages from source to destination through a network of point-to-point connections where the hosts (including the routers) are mobile and thus the connections between hosts are constantly changing [39, 23, 40, 18, 12, 13]. These routing algorithms tend to be complicated and to utilize substantial memory space (code and data), CPU time, and network bandwidth just to maintain the routing information, in addition to what the actual applications utilize.

Work has also been done on multicasting and broadcasting messages in an ad hoc network. Again, this work has focused on routing algorithms for delivering messages to certain specified hosts (multicast) or all hosts (broadcast) through a network of point-to-point connections, where the hosts are mobile and the connectivity changes constantly [2, 35, 48, 31, 50]. Work has also focused on *reliable* multicast and broadcast algorithms which ensure either that all intended destinations receive each message (in the same order, for some algorithms), or that none do [37, 38, 9]. All these algorithms require memory space, CPU time, and network bandwidth to maintain group membership and to enforce reliable message delivery and ordering guarantees.

M2MI and M2MP take a fundamentally different approach. Rather than trying to make address-based networking and routing work in an ad hoc mobile environment, M2MP eliminates the device addresses and groups altogether. Instead, all messages go to all devices within the proximal area (taking advantage of the wireless medium's inherent broadcast nature), and each device decides based on the message's contents whether and how to process the message. Also, M2MP does not guarantee reliable message delivery, error recovery being handled if necessary at higher levels in an application-specific fashion. When the device addresses, groups, and delivery guarantees vanish, so do the memory space, CPU time, and network bandwidth needed for the routing, group maintenance, and reliable delivery algorithms. This drastically simplifies M2MP, making it more attractive for small battery powered devices.

A potential problem with M2MI is a *broadcast storm* [35] where one device broadcasting a message causes other devices to broadcast messages, causing further broadcasts, and so on, leading to contention for the medium and diminished throughput. This problem was observed with *correlated* broadcasts resulting from broadcast-based flood routing. Consequently, M2MI-based applications must be designed to avoid correlated broadcasts.

8.2 Remote Method Invocation

Invocation of methods on remote objects is a well-established technique for constructing distributed systems, realized in distributed object systems like CORBA [36] and Java RMI [42]. Such systems use sending and receiving proxy objects (also called *stubs* and *skeletons*) to translate a method call to a message and back again. Typically, the proxy classes are compiled ahead of time from an interface definition file (as in CORBA) or from the actual Java interface (as in Java RMI). The proxy classes are then installed on all devices participating in the distributed application. Java RMI alternatively lets proxy classes be downloaded from a codebase server at run time, eliminating the need to install the proxy classes during application deployment.

While remote method invocation is indeed useful, existing distributed object system implementations have two drawbacks. First, pre-compiling and deploying the proxy classes in addition to the regular application classes entails additional effort and more opportunities for making mistakes. With Java RMI, if dynamic proxy downloading is used, a codebase (HTTP) server must be provided, various system properties must be set to point to the codebase URL, and a security policy must be put in place to permit connecting to the codebase server. Judging from the frequent pleas for help on RMI-related message boards, many people have trouble getting all this set up correctly. Also, using codebase URLs is problematic in an ad hoc networking environment where there are no predetermined host addresses and where there may not even be any host that can act as a codebase server.

The second drawback is that downloaded code, including downloaded RMI proxy code, poses a major security risk. While the Java virtual machine and security manager defend against some kinds of attacks, they do not defend against others. For example, downloaded code can mount a *denial of service attack* that crashes the system by allocating all available memory or spawning too many threads [32].

Downloaded code can be digitally signed, and the code can be prevented from executing unless it has a valid signature from a trusted source. However, the signature only verifies who created the code, not whether the code is benign. The signature may not even verify who created the code if the signing computer has been compromised [44]. Trusting downloaded code is especially problematic for a *device* that is expected never to "crash."

While using the same proxy-based technique as existing remote method invocation systems, with the handles and the method invokers taking the roles of the sending and receiving proxies, M2MI avoids the existing systems' deployment and security drawbacks. By synthesizing the M2MI proxy classes directly in the devices where they are used, proxy precompilation, codebase servers, and proxy class downloading are all eliminated. This simplifies M2MI-based application development and deployment, especially in an ad hoc networking environment. Since the M2MI layer synthesizes its own proxies, it can ensure that the proxies do only what they're supposed to do and not anything malicious — without needing to place trust in a code signer.

8.3 Distributed Systems Architecture

Figure 21 shows the design centers of several distributed systems architectures compared to the design center of M2MI. Each architecture is classified along three dimensions: whether the architecture is based on centralized servers; whether the hosts or devices are configured with each other's addresses ahead of time or discover each other dynamically at run time; and the communication patterns among the hosts or devices, one-to-one, one-to-many, or many-to-many.

The *client-server* architecture is based on a central server whose address (or URL) must be known ahead of time. Most client-server systems use one-to-one communication (e.g. email, web browsing); some use one-to-many communication (e.g. webcasting). While collaborative applications can be and have been built using a client-server architecture, a collaborative application's many-to-many communication pattern doesn't match the client-server architecture's design center. As a result, the application

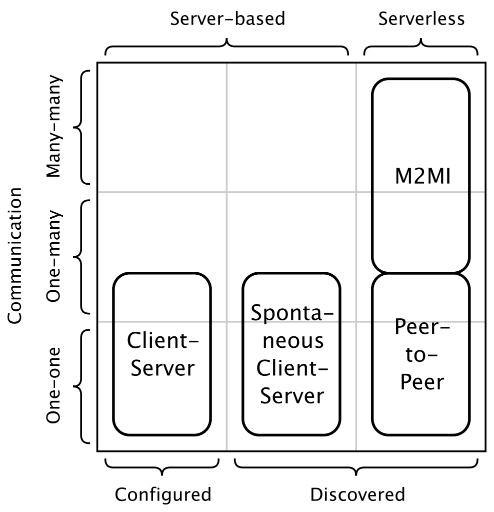


Figure 21: Design centers of distributed systems architectures

tends to communicate in a "star" pattern where each user's device sends messages to the server and the server then copies the messages to the other devices. In a proximal network with a broadcast medium, sending a separate copy of each message to each device wastes network bandwidth. Also, if the server goes down or becomes inaccessible, the application can no longer operate, even though the devices can communicate with each other directly. Finally, needing to know the server's address ahead of time is problematic in an ad hoc network.

The *spontaneous client-server* architecture eliminates the need for preconfigured addresses by providing a discovery mechanism. Jini Network Technology [1] is a good example. A lookup service runs on one or more server hosts. Clients and services discover the lookup service using a multicast protocol. Services upload their own proxy objects to the lookup service. Clients download the desired service proxy objects from the lookup service. Clients then invoke methods on the service proxy objects to communicate directly with the services. While this architecture does not require server addresses to be known ahead of time, applications are more complicated to develop because they must discover and interact with the lookup service in addition to their normal functions. Since the architecture still relies on central servers, there's still a mismatch for collaborative applications. Also, Jini in particular relies on downloaded code, which poses a security risk as discussed earlier.

Keeping the spontaneous discovery of services while eliminating the central servers results in a *peer-to-peer* architecture. M2MI is a peer-to-peer architecture oriented around one-to-many and many-to-many communication (although it also supports one-to-one communication). Unlike an application in a client-server architecture, an M2MI-based collaborative application runs collectively in all the participating devices, not on a central server. Thus, an M2MI-based application will not stop operating because a server crashed or became inaccessible. Like a spontaneous client-server architecture, M2MI discovers services dynamically rather than configuring servers' addresses statically. But unlike a spontaneous client-server architecture, M2MI has no central lookup services, and the application does not have to explicitly discover its partners before it can start interacting with them. Rather, the application just goes ahead and broadcasts M2MI method invocations, and whichever partners are out there will respond. This simplifies development and deployment of M2MI-based applications.

8.4 Collaborative Middleware

A number of middleware frameworks for building collaborative applications in ad hoc networks of mobile devices are under investigation. Some frameworks, such as Proem [29, 30] and JXTA [24], follow a *protocol-centric* paradigm in which a standard set of message formats (nowadays typically XML-based) is defined to let devices discover each other, exchange data and events, and otherwise interact with each other. Since the message formats are programming language neutral, applications can be written in different languages to run on heterogeneous platforms and still collaborate. In contrast, M2MI uses only one message "format," that of a method invocation, and overlays that with an object oriented abstraction in which applications interact by calling methods in interfaces rather than by sending messages. Since M2MI uses dynamic proxy synthesis which the Java platform makes possible, it would be difficult to run M2MI in a heterogeneous environment where some devices lack a Java virtual machine. This, however, is becoming increasingly less of a restriction as more and more devices, including handheld computers, personal digital assistants (PDAs), and cellphones, are shipped with Java.

Other frameworks follow a *data-centric* paradigm. In one.world [16], data are stored in tuples, and applications interact by reading and writing each other's tuples and sending each other events consisting of tuples. Lime [33] is based on "transiently shared tuple spaces" in which each device has a local tuple space, nearby devices merge their local tuple spaces into a shared global tuple space, and applications interact by reading and writing tuples in the shared space.

Different middleware frameworks offer different levels of abstraction. M2MI offers a low-level, method call oriented abstraction. A shared tuple space offers a high-level, data oriented abstraction. In fact, M2MI can be used to implement various high-level middleware frameworks. Applications can then be implemented using the high-level middleware or using M2MI directly. M2MI simplifies the development of high-level middleware frameworks as well as applications in a collaborative ad hoc environment.

9 Status and Future Work

The M2MI paradigm is a work in progress. The sections below describe the current status of M2MP, M2MI, M2MI-based collaborative systems, and security in the M2MI framework. Also described are plans for our ongoing work on M2MI.

9.1 Many-to-Many Protocol

The M2MP protocol has been defined and a prototype protocol stack has been written in Java. The prototype runs on desktop hosts. The prototype code, including a detailed description of the M2MP packet format, is available [26]. The prototype includes several channel implementations that use UDP datagrams to transport M2MP packets (see Section 7.5).

In our continuing work on M2MP, we plan to construct several additional channel implementations. One channel implementation, currently being developed, will transport M2MP directly over a wired Ethernet data link layer, eliminating the unnecessary protocol overhead of the UDP and IP layers in the prototype [20]. This channel implementation will then be extended to run over a wireless (802.11) Ethernet. Another channel implementation will transport M2MP over Bluetooth. The implementations will be written in Java, along with native code where necessary, and tested on a desktop host.

Once these implementations are working, we plan to migrate the M2MP protocol stack, including the shared memory layer, into the Linux operating system kernel. This will reduce the overhead and improve the performance of M2MP.

The way the M2MP packet format is presently defined, an adversary could disrupt a multi-packet M2MP message by injecting a packet with the correct next fragment number but a bogus message fragment. The M2MP layer would have no way of knowing that this packet is not authentic and would pass the bogus data up to the application. This attack is of especial concern in a wireless network, which is arguably easier for an intruder to access than a wired network.

To defend against a packet insertion attack, we plan to replace the checksum with a *message authentication code* (MAC), which is a one-way hash of the packet's contents that requires a key to compute or verify. Each packet uses a different randomly-chosen key. The key needed to verify a packet's MAC is carried in the *next* packet; a final empty packet carries the last key. To conduct a packet insertion attack, an adversary would have to determine the key from a packet's contents and MAC, so as to put the correct key in the next packet; but this is computationally infeasible. An initial version of this scheme has been implemented [4].

9.2 Many-to-Many Invocation

An initial prototype of M2MI has been written in Java. The prototype runs on desktop hosts. The prototype code is available [25]. The M2MI prototype uses the M2MP prototype [26] for messaging and the RIT Classfile Library [27] for dynamic proxy synthesis. It builds on an earlier prototype that used an offline proxy compiler [5].

In our continuing work on M2MI, we plan to port the RCL, M2MP protocol core, M2MP channel, and M2MI implementations to a PDA platform with Java capability and 802.11 or Bluetooth wireless connectivity. The porting effort may require redesigning and reimplementing the software to reduce the memory and CPU requirements to a level suitable for a small mobile wireless device. Once ported, we plan to test interoperation of M2MP and M2MI from PDA to desktop host and from PDA to PDA.

We also plan to develop tools to help develop and debug M2MI-based systems and to monitor and visualize M2MI-based systems during operation.

9.3 M2MI-Based Systems

Initial prototypes of several collaborative applications, including chat, IM, whiteboard, calendar, file sharing, and tuple space, have been constructed using M2MI. The prototypes run on desktop hosts.

From our initial investigations we are getting an inkling of a general paradigm for building collaborative systems using M2MI. Some elements of the paradigm are perceptible, such as participant discovery (see Sections 4.2 and 4.4), service discovery (see Section 4.5), multiple simultaneous groups (see Section 4.2), random selection of respondents to avoid broadcast storms (see Sections 4.3 and 4.7), and timeouts to recover from missing responses.

We plan to build up experience with and to codify the M2MI paradigm by developing a number of M2MI-based collaborative systems. The systems we plan to develop include:

- Full-featured chat and instant messaging, enabling spontaneous conversations in quiet spaces like libraries and museums
- Full-featured collaborative groupware, including presentation, shared whiteboard, note taking, document authoring by multiple simultaneous authors, file and information sharing, and calendar scheduling features
- Specialized applications for communication in noisy environments such as engine rooms, airfields, flight decks, meeting halls, and restaurants
- Multiplayer games
- Document system utilizing dynamically discovered print services, allowing users to find nearby printers and print from their devices wherever they happen to be
- Surveillance system utilizing dynamically discovered video cameras, allowing users to display images from nearby cameras wherever they happen to be
- Lightweight shared tuple space middleware framework like that of Lime [33]

Each system will be tested on a mixture of desktop and PDA platforms with wired and wireless connectivity.

As we gain experience building M2MI-based systems we plan to flesh out the collaborative system paradigm, devise reusable design patterns, and construct class libraries for building collaborative systems using the paradigm.

9.4 M2MI Security

Providing security within M2MI-based systems is an area for future work. As a starting point, we have identified these general security requirements:

- Confidentiality — Intruders who are not part of a collaborative system must not be able to understand the contents of the M2MI invocations.
- Participant authentication — Intruders who are not authorized to participate in a collaborative system must not be able to perform M2MI invocations in that system.
- Service authentication — Intruders must not be able to masquerade as legitimate participants in a collaborative system and accept M2MI invocations. For example, a client must be assured that a service claiming to be a certain printer really is the printer that is going to print the client's job and not some intruder.

While existing techniques for achieving confidentiality and authentication work well in an environment of fixed hosts, wired networks, and central servers, it is not clear which techniques would work well in an environment of mobile devices, wireless networks, and no central servers.

Consider, for example, an M2MI-based chat application that supports *closed sessions* where only certain users are allowed to participate. To achieve confidentiality, all the M2MI invocations can be encrypted using a key known only to the chat session members. Ideally, a user should be able to arrive where a closed chat session is taking place, prove that he or she is a member of the group (authentication), obtain the encryption key being used at that time (session key exchange), and start participating in the session. However, authentication and session key exchange systems such as Kerberos [28] rely on central servers that may not be available in an ad hoc device environment.

Building blocks such as the following may be more attractive for M2MI-based applications. Public key exchange protocols, such as Diffie-Hellman key exchange [10], do not require a central server. However, the parties in the exchange must be authenticated to prevent intruder-in-the-middle attacks. Authentication schemes based on zero-knowledge proofs of identity [11, 17, 41, 45] also do not require interacting with a central server. Furthermore, serverless techniques for proving group membership rather than individual identity, such as one-way accumulators [3], eliminate the need to maintain group membership lists on all devices and so may be more attractive in an ad hoc networking environment where all devices are not present all the time. Variations of such schemes based on elliptic curves are especially attractive for small devices, since to obtain a given level of security elliptic curve based algorithms typically require much less storage and processing than algorithms based on integers in a finite field [6].

To begin our investigation of M2MI security, we plan to conduct a literature search to identify cryptographic algorithms for achieving confidentiality and authentication that are suited for an environment of mobile devices, wireless networks, and no central servers. Where the existing algorithms are not well suited for that environment, we will define modified cryptographic algorithms that are better suited. To reduce memory and processing requirements in small devices, we will define elliptic curve based variants of the cryptographic algorithms where necessary. Finally, we will analyze how to extend the M2MI infrastructure to provide confidentiality and authentication.

10 Acknowledgments

Jim Waldo inspired the idea for M2MI when he said "Everyone that's out there, call this method" during a discussion about M2MP. Wendi Heinzelman suggested the digital photo sharing application in Section 4.7.

The following students at the Rochester Institute of Technology have helped us investigate the M2MI infrastructure and have built prototype M2MI-based applications: Adam Bazinet, Joseph Binder, Steve Button, Tom Chang, Dan Clark, Jonathan Coles, Frank Conover, Louie Gosselin, Kiran Hegde, John King, Brian Koponen, Kevin Mooney, Jeffrey Myers, Ravi Nareppa, Jim Papapanu, Tri Phan, Jacob Rigby, Girish Sarma, Anthony Stamp, Evan Teran, Hau San Si Tou, Ken VanderVeer, Merit Wilkinson, and Josh Zatulove.

We would like to thank Jeffrey Lasky, Amy Murphy, and the anonymous referees for their comments on earlier drafts of this paper.

This research was supported by grants from Sun Microsystems and Xerox Corporation.

References

[1] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification.* Addison-Wesley, 1999.

[2] S. Basagni, D. Bruschi, and I. Chlamtac. A mobility-transparent deterministic broadcast mechanism for ad hoc networks. *IEEE/ACM Transactions on Networking*, 7(6):799–807, December 1999.

[3] J. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology — EUROCRYPT '93, Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pages 274–285, May 1993.

[4] J. Binder, J. King, K. Mooney, and M. Wilkinson. Ad hoc wireless networks security system summary. Research seminar class project report, Rochester Institute of Technology, Rochester, NY, May 2002.

[5] H.-P. Bischof. Many-to-Many Invocation Compiler. `http://www.cs.rit.edu/~anhinga/downloads/historical.shtml`.

[6] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.

[7] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course.* MIT Press, 1990.

[8] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 44–57, December 1993.

[9] D. M. Chiu, M. Kadansky, J. Provino, J. Wesley, H.-P. Bischof, and H. Zhu. A congestion control algorithm for tree-based reliable multicast protocols. Technical Report TR-2001-97, Sun Microsystems, June 2001. `http://research.sun.com/nova/cgi-bin/smli_tr-2001-97.pdf`.

[10] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[11] U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988.

[12] J. J. Garcia-Luna-Aceves and M. Spohn. Bandwidth-efficient link-state routing in wireless networks. In C. E. Perkins, editor, *Ad Hoc Networking*, pages 323–350. Addison-Wesley, 2001.

[13] J. J. Garcia-Luna-Aceves and M. Spohn. Transmission-efficient routing in wireless networks using link-state information. *Mobile Networks and Applications*, 6(3):223–238, June 2001.

[14] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[15] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[16] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall. Programming for pervasive computing environments. Technical Report UW-CSE-01-06-01, University of Washington, Department of Computer Science and Engineering, June 2001. `http://one.cs.washington.edu/papers/tr01-06-01.pdf`.

[17] L. Guillou and J. Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *Advances in Cryptology — EUROCRYPT '88, Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pages 123–128, May 1988.

[18] Z. J. Haas and M. R. Pearlman. ZRP: a hybrid framework for routing in ad hoc networks. In C. E. Perkins, editor, *Ad Hoc Networking*, pages 221–253. Addison-Wesley, 2001.

[19] T. Hastings, R. Herriot, R. deBry, S. Isaacson, and P. Powell. Internet Printing Protocol/1.1: Model and Semantics. Internet Request for Comments (RFC) 2911, September 2000.

[20] K. Hegde. M2MP over Ethernet. Master's thesis, Rochester Institute of Technology, Rochester, NY, 2002. In progress.

[21] Internet Engineering Task Force. IP Routing for Wireless/Mobile Hosts (mobileip) Working Group. `http://www.ietf.org/html.charters/mobileip-charter.html`.

[22] Jini Printing Working Group, A. Kaminsky, editor. Jini Print Service API Draft Standard 1.0. `http://print.jini.org/`, May 2000.

[23] D. B. Johnson, D. A. Maltz, and J. Broch. DSR: the Dynamic Source Routing protocol for multihop wireless ad hoc networks. In C. E. Perkins, editor, *Ad Hoc Networking*, pages 139–172. Addison-Wesley, 2001.

[24] Project JXTA. `http://www.jxta.org/`.

[25] A. Kaminsky. Many-to-Many Invocation Library. `http://www.cs.rit.edu/~anhinga/m2mi.shtml`.

[26] A. Kaminsky. Many-to-Many Protocol Library. `http://www.cs.rit.edu/~anhinga/m2mp.shtml`.

[27] A. Kaminsky. RIT Classfile Library. `http://www.cs.rit.edu/~anhinga/rcl.shtml`.

[28] J. Kohl and C. Neuman. The Kerberos network authentication service (v5). Internet Request for Comments (RFC) 1510, September 1993.

[29] G. Kortuem, S. Fickas, and Z. Segall. Architectural issues in supporting ad-hoc collaboration with wearable computers. In *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing at the 22nd International Conference on Software Engineering*, June 2000. `http://www.cs.washington.edu/sewpc/papers/kortuem.pdf`.

[30] G. Kortuem, J. Schneider, D. Preuitt, T. G. C. Thompson, S. Fickas, and Z. Segall. When peer-to-peer comes face-to-face: Collaborative peer-to-peer computing in mobile ad hoc networks. In *Proceedings of the 2001 International Conference on Peer-to-Peer Computing (P2P2001)*, August 2001. `http://www.cs.uoregon.edu/research/wearables/Papers/p2p2001.pdf`.

[31] S.-J. Lee, W. Su, and M. Gerla. Wireless ad hoc multicast routing with mobility prediction. *Mobile Networks and Applications*, 6(4):351–360, August 2001.

[32] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business with Mobile Code.* John Wiley & Sons, 1999.

[33] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'01)*, pages 524–533, April 2001.

[34] National Institute of Standards and Technology. Digital signature standard. NIST FIPS PUB 186, May 1994.

[35] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99)*, pages 151–162, August 1999.

[36] Object Management Group. The common object request broker: Architecture and specification, revision 2.4.1, November 2000.

[37] E. Pagani and G. P. Rossi. Reliable broadcast in mobile multihop packet networks. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '97)*, pages 34–42, September 1997.

[38] E. Pagani and G. P. Rossi. Providing reliable and fault tolerant broadcast delivery in mobile ad-hoc networks. *Mobile Networks and Applications*, 4(3):175–192, October 1999.

[39] C. E. Perkins and P. Bhagwat. DSDV routing over a multihop wireless network of mobile computers. In T. Imielinski and H. F. Korth, editors, *Mobile Computing*, pages 183–206. Kluwer Academic Publishers, 1996.

[40] C. E. Perkins and E. M. Royer. The ad hoc on-demand distance-vector protocol. In C. E. Perkins, editor, *Ad Hoc Networking*, pages 173–219. Addison-Wesley, 2001.

[41] J. Quisquater, L. Guillou, and T. Berson. How to explain zero-knowledge protocols to your children. In *Advances in Cryptology — EUROCRYPT '89, Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pages 628–631, April 1989.

[42] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling state in the Java system. *Computing Systems*, 9(4):291–312, Fall 1996.

[43] R. Rivest. The MD5 message-digest algorithm. Internet Request for Comments (RFC) 1321, April 1992.

[44] B. Schneier. Why digital signatures are not signatures. `http://www.counterpane.com/crypto-gram-0011.html`, November 2000.

[45] C. Schnorr. Efficient signature generation for smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[46] Sun Microsystems. Java Object Serialization Specification. `http://java.sun.com/j2se/1.4/docs/guide/serialization/index.html`.

[47] Sun Microsystems. Java Print Service Specification. `http://java.sun.com/j2se/1.4/docs/guide/jps/index.html`.

[48] J. E. Wieselthier, G. D. Nguyen, and A. Ephremides. Algorithms for energy-efficient multicasting in static ad hoc wireless networks. *Mobile Networks and Applications*, 6(3):251–263, June 2001.

[49] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.

[50] S.-M. Yoo and Z.-H. Zhou. All-to-all communication in wireless ad hoc networks. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 180–181, March 2001. `http://webster.cs.uga.edu/~jam/acm-se/review/abstract/syoo.ps`.

A Chat Application Source Code

Figure 22 gives the Java source code for the M2MI-based chat application of Section 4.1 (the one with omnihandles and a single chat session).

Class `ChatDemo` is the main program. After initializing the M2MI layer,[3] the main program creates a chat UI object and a chat object. These objects do all the work.

The chat UI object, class `ChatFrame`, provides a simple graphical user interface to display the messages in the chat log and to let the user enter and send a new chat message. The chat UI object provides an operation to register a chat frame listener object (interface `ChatFrameListener`). The source code for class `ChatFrame` is omitted since it has nothing to do with M2MI.

Interface `ChatFrameListener` specifies the interface for a chat frame listener object. Whenever the user sends a new chat message, the chat UI object calls the `send` method on its registered chat frame listeners, passing in the chat message text.

Interface `Chat` is the target interface for M2MI invocations on the exported chat objects.

Class `ChatObject` is the exported chat object, which implements interfaces `ChatFrameListener` and `Chat`. When constructed, the chat object is given the chat UI object and the user name. The chat object registers itself with the chat UI object as a chat frame listener. The chat object also exports itself to the M2MI layer as target interface `Chat`. Finally, the chat object obtains an omnihandle for interface `Chat` from the M2MI layer.

When the user sends a new chat message, the chat UI object calls the chat object's — that is, the chat frame listener's — `send` method. (This is a normal method call, not an M2MI invocation.) The chat object prepends the user name to the message text and calls `putMessage` on the omnihandle for interface `Chat`.

The omnihandle invocation causes every chat object's `putMessage` method to be executed. Each chat object calls a method in its corresponding chat UI object to add the chat message to the chat log. (This is a normal method call, not an M2MI invocation.)

[3]The argument is a globally unique address for the M2MI layer that would typically be the host's network interface's MAC address.

```
public class ChatDemo
   {
   public static void main
      (String[] args)
      {
      M2MI.initialize (1234L);
      ChatFrame theChatFrame = new ChatFrame();
      ChatObject theChatObject =
         new ChatObject (theChatFrame, args[0]);
      }
   }

public interface ChatFrameListener
   {
   public void send
      (String line);
   }

public interface Chat
   {
   public void putMessage
      (String line);
   }

public class ChatObject
   implements ChatFrameListener, Chat
   {
   private ChatFrame myChatFrame;
   private String myUserName;
   private Chat allChats;

   public ChatObject
      (ChatFrame theChatFrame,
       String theUserName)
      {
      myChatFrame = theChatFrame;
      myUserName = theUserName;
      myChatFrame.addListener (this);
      M2MI.export (this, Chat.class);
      allChats = (Chat) M2MI.getOmnihandle
         (Chat.class);
      }

   public void send
      (String line)
      {
      allChats.putMessage
         (myUserName + "> " + line);
      }

   public void putMessage
      (String line)
      {
      myChatFrame.addLineToLog (line);
      }
   }
```

Figure 22: Chat application source code

Challenging Encapsulation in the Design of High-Risk Control Systems

Daniel Dvorak
JPL / California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
1-818-393-1986
Daniel.Dvorak@jpl.nasa.gov

ABSTRACT

In the hardware/software design of control systems it is almost an article of faith to decompose a system into loosely coupled subsystems, with state variables encapsulated in device and subsystem objects. The engineering advantages of such an approach are so attractive that it is sometimes applied inappropriately, yielding a design that hides a tangle of special-case subsystem-to-subsystem couplings behind a façade of modular decomposition. The limitations of a device/subsystem architecture become apparent in the design of high-risk control systems—such as nuclear power plants and planetary rovers—where the world is full of physical side-effects that have little "respect" for conventional subsystem boundaries. Here, the very notion of decomposition by subsystem, and its attendant state encapsulation, actually complicates the design. Fundamentally, there is a clash between a device-subsystem-object metaphor and the laws of physics. A more appropriate architectural approach is to acknowledge the underlying physics and to elevate the concepts of *state* and *models* to first-class design elements that are *not* encapsulated within subsystem objects.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures, information hiding.*

General Terms

Design.

Keywords

Encapsulation, isomorphism, state, model, control system, design, architecture.

1. INTRODUCTION

Software systems vary enormously in the extent to which they interact with the physical world and deal with its subtleties. At one extreme are enterprise applications such as management information systems that deal primarily in a tidy world of data management, queries, and reporting. Such systems are typically deployed in an environment of plentiful resources — plenty of data storage, network throughput, electrical power, and air conditioning. In such an environment most physical side effects can be safely ignored. For example, powering up a disk drive in a server room consumes power, generates heat, and imparts a rotational torque on the disk drive assembly. These are real physical effects, but we can safely ignore them as irrelevant side effects when the resources that they affect are virtually unlimited. In this situation, power and air conditioning and rotational inertia are all virtually unlimited.

At the other extreme are resource-limited robots such as unmanned spacecraft and Mars surface rovers. Since the amount of mass launched into space is a major cost driver for space missions, these systems are engineered to carry only enough resources to accomplish mission objectives, plus a small margin. Mission activities must be designed to operate within tightly engineered constraints on electrical power, battery energy, non-volatile memory, communication link throughput, and many other resources. For example, turning on a camera to take pictures draws from a limited power budget, consumes non-volatile memory to store images, and requires the rover basebody to be pointed appropriately. This activity uses precious resources that are then *not* available to other activities. The net result is that in a resource-limited system many physical side effects become *non-*negligible and therefore must be consciously managed; designs become more complex because the couplings are more numerous and often cross conventional subsystem boundaries.

This paper compares two architectures with respect to their suitability for resource-limited control systems. One architecture is device/subsystem-oriented, having objects associated with hardware units, such as drive motors and camera, plus objects associated with traditional engineering subsystems such as electrical, thermal, and navigation subsystems. This architecture encapsulates state variables inside such objects—objects that logically seem to "own" those state variables. The other architecture is state/model-oriented, having first-class objects associated with physical states, such as camera temperature and batter energy, plus objects associated with models of physical couplings, such as the effect of a heater on power and the effect of temperature on a sensor measurement. Despite the appeal of decomposition by subsystem, the structure of a device/subsystem architecture depends on an assumption of loose coupling that simply doesn't hold in the realm of resource-limited systems.

17th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, November 4-8, 2002, Seattle, Washington.

Such systems must manage numerous tight couplings of the real world. Dealing with this in a disciplined way demands an architecture that acknowledges and adequately represents the underlying physics of the world.

2. DEVICE/SUBSYSTEM ARCHITECTURE

Most physical systems that people deal with on a daily basis are designed as a composition of modular subsystems that interact in a few obvious and easily controlled ways. Such systems are easier to understand, easier to monitor and control, and easier to diagnose when faulty. For example, the different subsystems of a modern home—electrical, heating/cooling, plumbing, telephone, and cable—are relatively independent of each other, with only a few important forms of coupling. In truth, there are *many* physical couplings among these subsystems, but most are negligible. For example, the electrical subsystem is a source of electromagnetic interference to the telephone and cable subsystems, but the signal-to-noise ratio is high enough that the effects can be ignored. Similarly, the circulation of hot water in the plumbing system affects the heating/cooling system, but the effect is negligible compared to the heating/cooling subsystem's ability to notice and compensate. These and other physical effects are negligible because the system has abundant resources: plenty of electrical power, abundant heating/cooling capacity, large thermal mass, substantial electrical and thermal insulation, plenty of electromagnetic shielding, etc.

In such an environment of plentiful resources, control system software designers can appropriately treat subsystems as largely independent, with state variables encapsulated in the subsystem or device object that has the dominant effect or "ownership" of that state. For example, hot water temperature could be encapsulated with the water heater object since the water heater has the dominant effect on that state. Of course, there are some subsystem couplings that cannot be ignored because they have intentional effects or major side effects. For example, an electric hot water heater depends on power from the electrical subsystem in order to operate, so this one-way dependency must appear somewhere in the system logic. This kind of subsystem coupling is so simple to describe and so few in number that it's typical for a software designer to express it on a case-by-case basis, often in the logic of one or two operations of a class. Figure 1 shows an ideal decomposition by subsystem, where the only couplings are those between a child subsystem and its parent subsystem.

2.1 Example: Home Heating System

An illustrative example of the subsystem/device approach to control systems is the "Smalltalk Home Heating System" described by Booch [1, chapter 8]. As Booch notes, the home heating system naturally decomposes into relatively independent subproblems. The heating system contains top-level objects of a furnace, heat flow regulator, operator interface, and a home consisting of multiple rooms. Each room has a current temperature sensor, a desired temperature sensor, a room occupancy sensor, and a water valve.

Note that top-level software objects model the obvious physical elements of a home heating system, and that makes sense as long as most or all of the couplings exist *within* containment and/or attachment relationships. For example, in this system there is nothing outside of a room that affects the measurements from the current temperature sensor, so it makes sense to encapsulate the current temperature state within the room or its temperature sensor.

In summary, a device/subsystem architecture is appropriate for many everyday control systems because they exhibit simple subsystem couplings. Such an architecture is extremely attractive to most designers because software structure reflects hardware structure and follows familiar subsystem-oriented decomposition. The only danger, which this paper explores, is that designer familiarity with this appealing architecture can lead to its use in applications where physical couplings are complex and have little "respect" for the hardware structure and subsystem boundaries.

2.2 Couplings Due to Physics

The limitations of a device/subsystem architecture aren't apparent in "everyday systems" in much the same way that the limitations of Newtonian physics aren't apparent in everyday experiences, where velocity is a tiny fraction of the speed of light. In the case of the device/subsystem architecture the limitations start to appear in complex systems where the everyday assumptions of loose coupling simply don't hold. To illustrate this point, consider a common, everyday device: a battery-powered electronic thermostat.

The job of a thermostat is to regulate temperature by sensing the temperature and by issuing on/off control actions to maintain temperature within a specified range. The thermostat itself is a self-contained hardware unit with two simple couplings: it senses ambient temperature and it opens and closes electrical contact between two terminals. Since we're talking about software design, let's assume that this electronic thermostat has a microprocessor running software that performs the temperature regulation. This looks like a perfect example of a hardware device that is loosely coupled with respect to an overall heating system, and indeed it is if that's an everyday home heating system.

Now consider the same task of temperature regulation, but in a very different environment: a Mars rover. Two things make this system very different: electrical power is extremely limited due to battery and solar panel limitations, and fault protection is a major concern since there are no repair technicians on Mars (as far as we know ☺). A consequence of limited power is that temperature regulation cannot be treated as an isolated activity; it has to be coordinated with other power-consuming activities such as driving, communication, and science instrument usage. Thus, as much as we would *like* to think of temperature regulation as the duty of a self-contained thermostatic unit, it can't be designed that way when it relies on the availability of an extremely limited resource.

In a similar way, designing for fault protection reveals other flaws with the idea of a loosely coupled thermostat. For example, if the temperature sensor fails, how do you estimate ambient temperature? Well, thanks to physics, there are other sources of evidence about temperature that can and should be used. For example, the recent history of the heater's on/off state provides important evidence about heating. The position of the Sun and its heating effect can be predicted with a thermal model. Likewise, power usage of nearby instruments and other devices has a heating effect that can be predicted, provided that those power states are accessible. Now, our once-simple thermostat has a lot more couplings and a lot more to think about. Also, suppose that the heater fails. How then can you control temperature? Well, again, physics offers the clues. One way may be to turn on instruments solely for their heating effect, subject to power availability. Another way is to reschedule activities that depend on temperature regulation to occur during mid-day on Mars when solar heating is at a maximum. Again, our once-simple thermostat is being asked to exercise control that goes far beyond its original role in the "thermal subsystem". The very concept of a self-contained thermostat is falling apart because its design rests on an assumption of loose coupling that simply doesn't hold in this more complex domain.

3. COUPLING AND URGENCY

In a significant book that analyzed accidents in complex systems such as Apollo 13 command module and the Three Mile Island nuclear power plant, Perrow [4] summarized the inherent risks in different types of systems using a Coupling/Urgency chart[1], as shown in Figure 2. The horizontal axis of *coupling* ranges from linear to complex. Linear couplings are those in expected and familiar production or maintenance sequence, and those that are quite visible even if unplanned. Complex couplings are those of unfamiliar sequences, or unplanned and unexpected sequences, and either not visible or not immediately comprehensible. As Stevens et al explain, "strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules." [5]

The vertical axis of *urgency* ranges from low to high. Low urgency systems can incorporate shocks and failures and pressures for change without destabilization. Low urgency systems tend to have ambiguous or perhaps flexible performance standards. High urgency systems have more time-dependent processes: they cannot wait or stand by until attended to. Reactions, as in chemical plants, are almost instantaneous and cannot be delayed or extended.

The placement of systems on this chart is based entirely on Perrow's subjective judgments because there is no standard way to measure the two variables of coupling and urgency. Nonetheless, the chart offers a useful qualitative comparison of different kinds of systems, and the history of system-level accidents supports his finding that complex couplings and high urgency make systems more prone to mishaps.

This paper focuses on coupling as an architectural driver. While the time-sensitive aspect of urgency is certainly important in system design, it is not key to the main point of this paper.

3.1 Coupling in Space Missions

As Figure 1 shows, space missions exist in the upper right quadrant of complex coupling and high urgency. Systems in this quadrant are at the highest risk for system-level accidents because they are harder to design and operate correctly.

Space missions exhibit complex coupling because many resources are severely limited. Some limitations, such as battery energy and solar panel power production, are due to the high cost of launching mass into space. Smaller batteries and smaller solar panels help reduce that cost. Other limitations such as processing speed and instrument usage ensue from the power limitation; running the processor at a lower clock rate and using one instrument at a time reduces power consumption. Still other limitations arise from the vast distances of outer space, where data communication rates fall as the square of the distance between transmitter and receiver. That means that it takes a long time to transmit data, and that activity typically precludes other activities while the antenna is carefully pointed at a moving target (such as Earth). Antenna pointing usually depends on basebody pointing, which is another managed resource.

Coupling occurs in many ways, including coupling through shared busses, structure, thermal proximity, grounding, environment, and so on. Most couplings are a direct consequence of system-level design, such as an instrument that will be damaged if it is in the wrong mode when thrusters fire. In addition, some couplings result from hardware design flaws that are discovered too late to fix, prior to launch. Examples include motor commands that cause processors to do a power-on-reset and communication busses that lock up when the wrong combination of units is active. To exaggerate just a bit, in a resource-limited system "everything affects everything".

3.2 Problems of Device/Subsystem Approach

The main problem in applying a device/subsystem architecture to resource-limited systems is that the architecture provides no leverage in dealing with the many non-negligible inter-subsystem couplings. Each such coupling has to be handled as a special case, leading to a tangle of subsystem-to-subsystem interactions hidden behind a façade of modular decomposition. In effect, the original architecture becomes an appealing fiction.

If a system is to be controlled efficiently then these couplings must be taken into account, for otherwise some less efficient scheme would have to be used in a loosely-coordinated manner. An example of the latter in spacecraft operations has been to reserve generous resource margins to ensure that a desired activity succeeds in spite of its side effects on limited resources. For example, operators may hold a 25% power margin above and beyond the predicted needs of the planned activities. This conservative strategy is understandable given the unforgiving nature of outer space, but it causes a spacecraft or rover to be significantly underutilized relative to its potential.

Interestingly, the practice of iterative development coupled with a subsystem decomposition can lead a project into a kind of architectural trap (though it should be understood that this is a secondary issue). Iterative development enables a team to demonstrate early progress and gain confidence by building a solution to a simplified problem, and then iterating to extend and refine the design. Unfortunately, the initial simplifying assumptions may be quite compatible with subsystem decomposition, leading the project into an architecture that fails to help when it is needed most—late in the development lifecycle when high-fidelity behavior must be achieved. As new iterations require higher fidelity behavior, new couplings that cross device and subsystem boundaries must be handled. Each one by itself is a small blemish on an otherwise tidy architecture, but achieving true high-fidelity behavior for the final delivery can render the original architecture largely irrelevant.

[1] To more closely match computer science terminology, this paper uses the terms 'coupling' and 'urgency' in place of Perrow's 'interaction' and 'coupling', respectively.

4. STATE/MODEL ARCHITECTURE

If a device/subsystem architecture is problematic for resource-limited systems, then what's a better approach? At a minimum, it has to be an approach that facilitates a software description of physical interactions, since management of those interactions is a dominant force in the design of resource-limited systems. It has to describe how things affect each other in the physical world, and this is exactly the role of *models* in the state/model architecture. As described below, there are three kinds of effects to model: measurement effects, command effects, and state effects. These models exist to support state estimation and state control, described later.

Just as the notion of *model* is elevated to a first-class entity, so also is the notion of *state variable*. Many state variables have no obvious encapsulating home within a subsystem-oriented architecture because many physical influences on their values have no "respect" for boundaries drawn by subsystem designers. Such state variables must stand on their own, apart from subsystems. The notion of state used here is broad, including many kinds of physical quantities such as temperature, pressure, switch position, device health, and position of one body relative to another. Together, state variables and models provide the means for describing physical interactions in software.

4.1 State Variables

In the realm of control systems, state variables are what system engineers identify and what operators monitor and control. Example states include the on/off position of a power switch and the orientation of a spacecraft. "State knowledge" always has associated uncertainty because sensors are imperfect, as are our models of how things work. Explicit representation of uncertainty enables estimators to be honest about the evidence and controllers to be cautious during periods of high uncertainty

4.2 Models

4.2.1 Measurement Effects Models

Sensors are hardware devices that produce measurements. Most real-world sensors are designed to measure a particular physical quantity, but they inadvertently and/or unavoidably measure other quantities. For example, a voltage sensor will produce a voltage measurement, but its value may be sensitive to temperature and magnetic field strength. Its value is also sensitive to its own calibration parameters of bias and scale factor. Finally, its value is affected by the sensor's health state, which may be in any of several failure modes.

A measurement model is a mapping from state(s) to measurement. In the example above, the voltage sensor's measurement model is a function of six states: voltage, temperature, magnetic field strength, sensor bias, sensor scale factor, and sensor health. Notice that temperature and magnetic field strength are *external* influences on the voltage measurements. Hence, this measurement model expresses two interactions that are independent of a subsystem hierarchy.

4.2.2 Command Effects Models

Actuators generate physical effects in response to commands. In addition to their intended effect, many actuators have unintended and/or unavoidable side effects. For example, a command to turn on a science instrument on a Mars rover has the desired effect of activating the instrument, but it also draws power from a limited supply, it causes localized heating that may affect other things (such as the voltage sensor mentioned previously), it may generate a magnetic field that interferes with another instrument, and it may start transmitting on the data bus, using up part of its limited capacity. Finally, the effects always depend on the actuator's health state, which may be in any of several failure modes.

A command effects model predicts the multiple effects of a command issued to an actuator in a given state. In this example the command effects model must predict the effect of a particular command on the values of five states: instrument activation state, battery power, nearby temperature, nearby magnetic field, and bus data rate. Notice that all of these effects, except for instrument activation, are *external* to the instrument. Hence, this model expresses four couplings that would violate an idealized subsystem hierarchy.

4.2.3 State Effects Models

In the physical world some states affect other states according to laws of physics and/or consequences of hardware design. For example, Boyle's ideal gas law expresses the relation between pressure state, volume state, and temperature state ($PV = nRT$). Similarly, the voltage drop across a resistor in an electrical circuit is a consequence of Ohm's law ($V=IR$). Likewise, the open/closed state of a valve affects flow state as well as both downstream and upstream pressure states.

A state effects model expresses such functional relations among states, and just as with measurement effects models and command effects models, the effects often span subsystem boundaries. Further, these are not necessarily just one-way effects; the ideal gas law describes a constraint that holds among multiple variables, any of which may be controllable or uncontrollable in a given system.

4.3 Estimators and Controllers

The three kinds of models described above provide a disciplined way of representing interactions that *must* be reasoned about in resource-limited systems. Accordingly, the architecture should elevate the concepts of state and models as first-class elements so that the numerous inter-subsystem couplings can be exposed and represented, not concealed through back-door device-to-device and subsystem-to-subsystem connections.

Such an architecture must perform state determination and state control *somewhere*, but in general it can't be done inside device or subsystem objects because they don't have sole 'ownership' of the states. As the preceding sections on models illustrated, for any given state there may be different measurements from different sensors that provide evidence about its value. Likewise, for any given state, there may be different commands to different actuators that can affect its value.

These simple facts suggest that *estimators* and *controllers* also need to be first-class architectural elements, distinct from the software objects for sensors and actuators and their aggregations. After all, if there are multiple sources of evidence about a state's value, there should be one entity that combines that evidence into an estimate. Likewise, if there are multiple ways of influencing the value of a state, there should be one entity that has overall responsibility for controlling that state.

Estimation and control are seen as distinct elements in this architecture and should *not* be combined, as is often the case in a subsystem approach. The simplest reason is clarity and correctness; it is easier to design, develop, and test two software modules where each has a single purpose than one module that tries to do two distinct things.

The job of an estimator is to interpret many sources of evidence—from measurements, commands, and state variables—given models of how things work. Evidence may be noisy, inconsistent, corrupted, and incomplete. In contrast, the job of a controller is to issue commands, as appropriate, in an attempt to influence the value of a state variable to satisfy a goal. Commands may have delayed effects and actuators may fail.

A second reason for separating estimation from control is more subtle; when the two tasks are combined, there is a temptation to shortcut the estimation process and never actually estimate the state to be controlled, but rather to modify flags and counters that the control logic "understands". This practice leads to systems that are hard for operators to monitor and understand because many key states are never explicitly estimated, and so the only way to understand them is to read the code.

4.4 Hardware Adapters

In this architecture the role of the hardware device object has been diminished relative to the device/subsystem architecture. Its main role now is to provide access to the hardware sensors and actuators. Estimators obtain measurements from sensors as inputs to the state estimation process, and controllers submit commands to actuators to influence physical state. In many cases, state variables that seem to be owned by a device should *not* be encapsulated in such objects because fault diagnosis reasoning within estimators and fault response logic within controllers often need access to such "internal" states.

4.5 Mission Data System

The architecture just described is that of the Mission Data System (MDS), a state and model-based architecture for resource-limited control systems, originally designed for unmanned spacecraft and planetary rovers [2]. The MDS architecture can be understood in terms of a few basic elements, as depicted in Figure 3.

- *State*. The MDS architecture is fundamentally state-based. States are what system engineers identify, what software engineers design and implement, and what operators monitor and control. Example states include the on/off position of a power switch and the orientation of a spacecraft. "State knowledge" always has associated uncertainty because sensors are imperfect, as are our models of how things work. Explicit representation of uncertainty enables estimators to be honest about the evidence and controllers to be cautious during periods of high uncertainty.

- *Models*. Much of what makes software different from mission to mission is domain knowledge about instruments, actuators, sensors, wiring, plumbing and many other things. By expressing such knowledge in inspectable models, apart from reusable software, the task of customizing MDS for a mission, then, becomes more a task of defining and validating models. Importantly, measurement models, command effects models, and state effects models provide an architectural basis for representing couplings.

- *Goals*. Goals are the basis for mission operations. A goal specifies operational intent as a constraint on the value of a state variable during a time interval. Importantly, a goal does not specify actions needed to accomplish it, thus leaving options open for autonomous control mechanisms. Goals enable operators to focus on *what* to accomplish rather than *how* to accomplish it. Active goals live in a goal network that specifies parent/child relationships and timing & ordering relationships.

- *State control*. State control encompasses the mechanisms devoted to goal achievement. This includes elaboration of a goal into subgoals, scheduling of goals on state timelines, time-based and event-based initiation of goal execution, delegation for real-time coordinated control, and hardware commanding.

- *State determination*. The task of estimating system state requires interpretation of many sources of evidence—such as measurements and commands—given a model of how things work. Evidence may be noisy, inconsistent, corrupted, and incomplete. State determination is a complicated enough job that it is deliberately separated from state control, thereby facilitating understandability, verification, and reuse.

5. RELATED WORK

In a 1995 joint study between NASA Ames and JPL known as the New Millennium Autonomy Architecture Prototype (NewMAAP) a number of existing concepts for improving flight software were brought together in a prototype form. These concepts included goal-based commanding, closed-loop control, model-based diagnosis, onboard resource management, and onboard planning. When the Deep Space One (DS-1) mission was subsequently announced as a technology validation mission, the NewMAAP project rapidly segued into the Remote Agent project [3]. In May 1999 the Remote Agent eXperiment (RAX) flew on DS-1 and provided the first in-flight demonstration of the concepts. The MDS project was established in April 1998 to define and develop an advanced multi-mission data system that unifies the flight, ground, and test elements in a common architecture. That architecture is shaped with the themes described in this paper, some of which were explored and refined by the RAX experience.

6. SUMMARY AND CONTRIBUTIONS

In the design of everyday control systems, the "divide and conquer" approach decomposes a system into loosely coupled subsystems that reflect traditional engineering disciplines such as power, thermal, navigation, telecommunication, science, etc. Each subsystem "owns" the estimation and control of particular states, so those state variables are encapsulated within the subsystem or its sub-subsystems, including proxy objects for devices that are considered to be part of the subsystem. This approach is workable—provided that the subsystems are loosely coupled—and is appealing because it supports a work breakdown according to engineering disciplines.

In contrast, high-risk control systems differ from everyday control systems in that the traditional subsystems are *not* loosely coupled, for two main reasons. First, in an environment where numerous activities compete for a share of limited resources, those activities must be coordinated in a way that is simply unnecessary when the resources are virtually unlimited. Second, in an environment where fault-tolerant control is a high priority, control decisions often must extend beyond the confines of a single subsystem. In short, the fundamental premise behind subsystem decomposition—loose coupling—does not hold, so a design based on such a decomposition would have to violate its own premise numerous times to deal with couplings that cross subsystem boundaries.

In designing for a high-risk control system, analysis of the physics of interactions suggests the shape of a more suitable architecture. States of the physical world clearly exist, but they do not owe their existence to a subsystem; they simply "are", and the job of the control system is to estimate their values as best as possible and control them as best as possible, even in the presence of faults. Estimation and control both depend on knowledge of how things work and how they fail, and that knowledge must be expressed somewhere as models of states, commands, and measurements.

The engineering contributions of this paper lie in five design principles of the state/model architecture: (1) state variables as a first-class elements, not subsystems; (2) explicit use of models to express the physics effects of couplings; (3) clean separation of state determination logic from state control logic; (4) explicit management of physical resources (power, memory, etc); and (5) the use of state constraints for operational control.

Designing for high-risk systems requires a paradigm shift from subsystem-oriented to state-oriented thinking. "Divide and conquer" must give way to "state analysis and physics modeling". Managing interactions is the key to good design in this domain, and if architecture is to be a help rather than a hindrance, it must facilitate representation and reasoning about such interactions.

It is not the intention of this paper to criticize state encapsulation or information hiding but rather to rethink what kinds of states are encapsulated in what kinds of classes. In a subsystem-oriented architecture the classes represent subsystems and devices, and the encapsulated states are seen as states that are wholly owned and/or wholly controlled by its subsystem. In a state-oriented architecture some classes represent and encapsulate individual physical states and have query, update, and notification operations for appropriate clients. Other classes represent such clients for state estimation, real-time control, and deliberative control. The lesson here, in the context of high-risk control systems, is that some state variables should *not* be encapsulated within subsystem objects because there is no single subsystem having full responsibility for the variable's value.

7. EPILOGUE

Engineering disasters can be great learning experiences. The 1930s design of the first Tacoma Narrows Bridge followed a popular trend toward lightness, structural grace, and flexibility. In fact, the original design had a 25 foot deep stiffening truss, but was later changed to an eight foot shallow plate girder, resulting in a much lighter bridge. Although the bridge was the epitome of artistry, it collapsed spectacularly in 1940 due to wind-induced vibrations because aerodynamic phenomena had not been adequately addressed in the design.

The same dangers of esthetics versus physics exist in software design, especially since the appearance of a design in UML diagrams (its esthetics) tends to be more visible to software engineers than the physics at play. Another esthetic is the appeal of a popular pattern, such as decomposition by traditional engineering subsystem; it seems reassuring since it has worked so well before. The fact that it clashes with the physics of interactions is sometimes hard to see because software is so malleable; it's always easy to add "one more interface" to accommodate a newly discovered need. The architectural end result becomes an appealing fiction: a tidy set of subsystems that hide a tangle of private, back-door interactions.

As software architects we must be careful about applying comfortable metaphors since they have the power to lead us astray. Object-oriented analysis is appealing because people can engage in anthropomorphic storytelling as a design strategy. That encourages secondary metaphors like 'ownership', which then map into subsystems and encapsulation. The fact that this approach works well in everyday control systems encourages architects to apply it to all control system problems. With such a mindset, it is hard to recognize when a new design problem is qualitatively different from previous successfully solved problems. The best antidote for this is an objective analysis of the phenomena in play and the system-wide couplings that must be managed; those are the keys to good design. Only after that is done should one consider architectural styles.

8. ACKNOWLEDGEMENTS

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. Robert Rasmussen, chief architect of the Mission Data System, originally identified the existence of complex interactions—and the need to manage them architecturally—as a key driver in the MDS state/model architecture. Erann Gat documented this driver in an early internal document and later helped to shape the MDS state architecture. In another internal document, Kirk Reinholtz, chief programmer of MDS, documented the key limitation of the subsystem/device architecture in resource-limited systems.

9. REFERENCES

[1] Booch, G. Object Oriented Design with Applications. Benjamin/Cummings Publishing, 1991.

[2] Dvorak, D., Rasmussen, R., Reeves, G., and Sacks, A. Software Architecture Themes in JPL's Mission Data System. Proceedings of the 2000 IEEE Aerospace Conference, Big Sky, Montana, March, 2001.

[3] B. Pell, D. Bernard, S. Chien, E. Gat, N Muscettola, P. Nayak, M. Wagner, B. Williams. An Autonomous Spacecraft Agent Prototype. Proceedings of the First Annual Workshop on Intelligent Agents, Marina Del Rey, CA, 1997.

[4] Perrow, C. "Normal Accidents: Living with High-Risk Technologies." Basic Books, 1984.

[5] Stevens, W., Meyers, G., and Constantine, L. Structured Design, in *Classics of Software Engineering*, Yourdon Press, 1979.

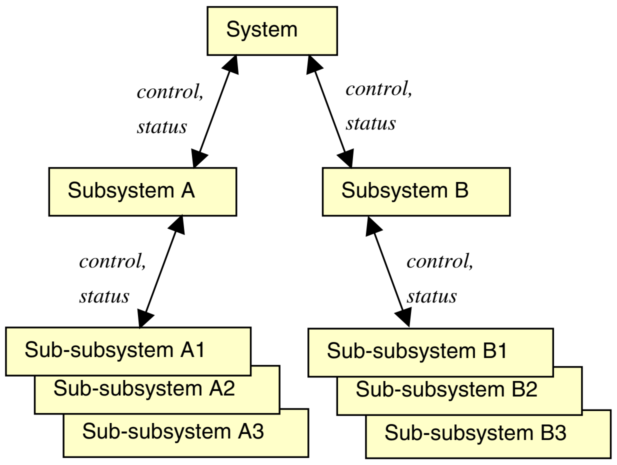


Figure 1. An architecture based on subsystem decomposition rests on an assumption of loose coupling, where interactions among subsystems are handled via hierarchical pathways of control and status. In such an architecture state variables are encapsulated within the object that has responsibility for its estimation and control.

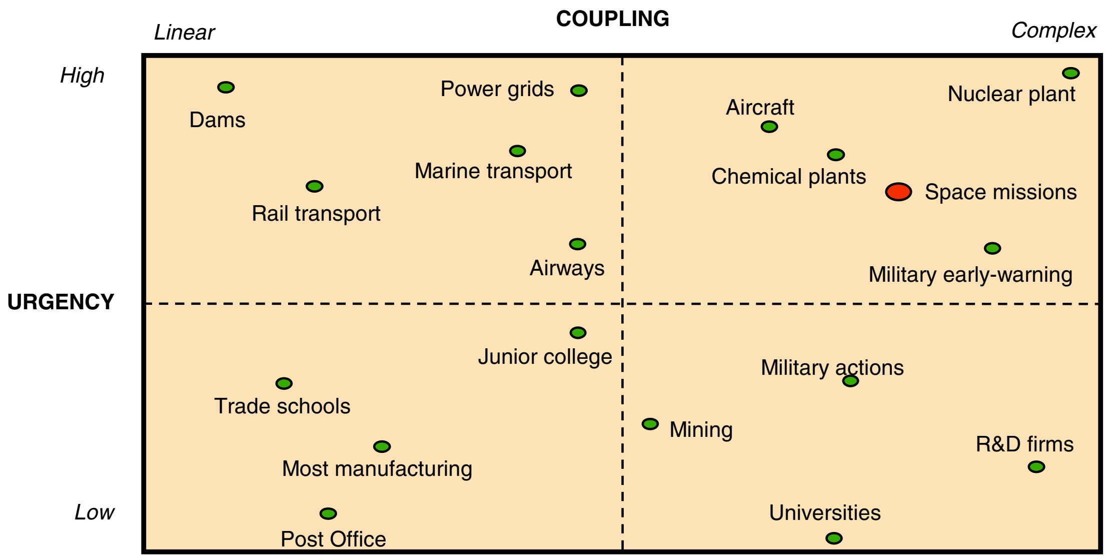


Figure 2. Systems that exhibit complex coupling and high urgency are considered high-risk because they are more prone to system accidents. This chart is due to Perrow [4].

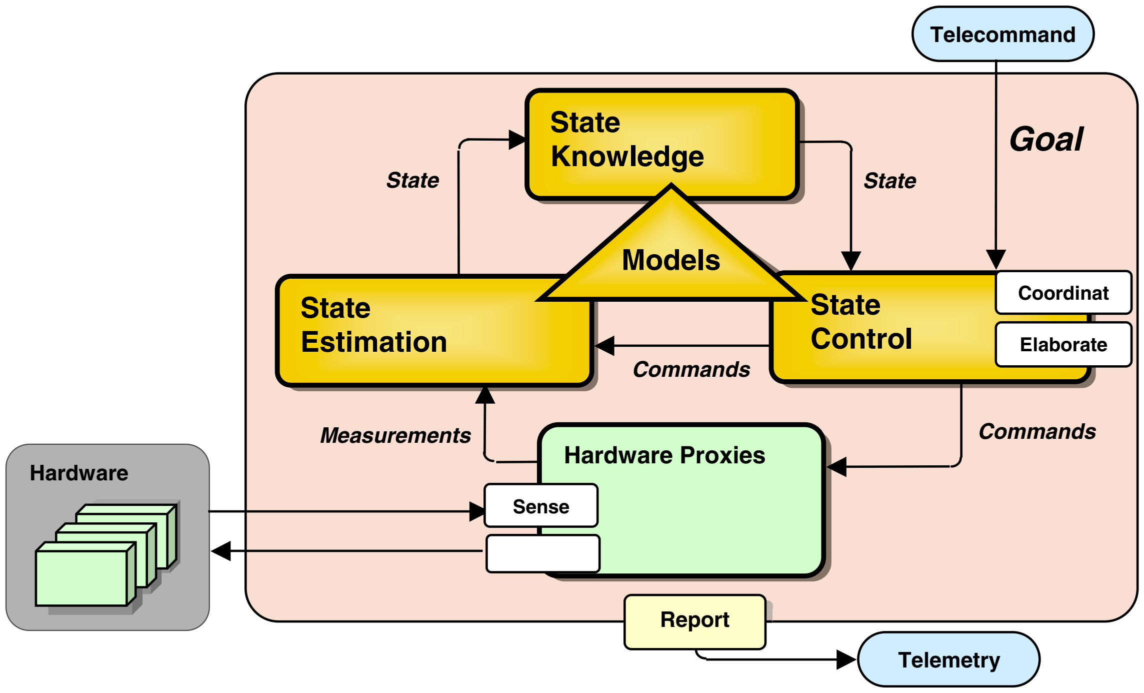


Figure 3. The state/model architecture of the Mission Data System emphasizes the central role of state knowledge and models, goal-driven operation, and separation of state determination from control.