Political Artifacts and Personal Privacy: The Yenta Multi-Agent Distributed Matchmaking System

by
Leonard Newton Foner

SB Electrical Engineering and Computer Science
Massachusetts Institute of Technology
June 1986

SM Media Arts and Sciences
Massachusetts Institute of Technology
June 1994

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in Partial Fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY
at the
Massachusetts Institute of Technology
June 1999

Signature of Author

Program in Media Arts and Sciences
April 30, 1999

Certified By

Pattie Maes
Associate Professor of Media Arts and Sciences
Program in Media Arts and Sciences

Accepted by

Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Political Artifacts and Personal Privacy: The Yenta Multi-Agent Distributed Matchmaking System

by
Leonard Newton Foner

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning, on April 30, 1999
in Partial Fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY
at the Massachusetts Institute of Technology

Abstract

Technology does not exist in a social vacuum. The design and patterns of use of any particular technological artifact have implications both for the direct users of the technology, and for society at large. Decisions made by technology designers and implementors thus have political implications that are often ignored. If these implications are not made a part of the design process, the resulting effects on society can be quite undesirable.

The research advanced here therefore begins with a political decision: It is almost always a greater social good to protect personal information against unauthorized disclosure than it is to allow such disclosure. This decision is expressly in conflict with those of many businesses and government entities. Starting from this premise, a multi-agent architecture was designed that uses both strong cryptography and decentralization to enable a broad class of Internet-based software applications to handle personal information in a way that is highly resistant to disclosure. Further, the design is robust in ways that can enable users to trust it more easily: They can trust it to keep private information private, and they can trust that no single entity can take the system away from them. Thus, by starting with the explicit political goal of encouraging well-placed user trust, the research described here not only makes its social choices clear, it also demonstrates certain technical advantages over more traditional approaches.

We discuss the political and technical background of this research, and explain what sorts of applications are enabled by the multi-agent architecture proposed. We then describe a representative example of this architecture---the Yenta matchmaking system. Yenta uses the coordinated interaction of large numbers of agents to form coalitions of users across the Internet who share common interests, and then enables both one-to-one and group conversations among them. It does so with a high degree of privacy, security, and robustness, without requiring its users to place unwarranted trust in any single point in the system.

Thesis Supervisor: Pattie Maes
Title: Associate Professor, Program in Media Arts and Sciences

This work was supported in part by British Telecom and Telecom Italia.

Political Artifacts and Personal Privacy:
The Yenta Multi-Agent Distributed Matchmaking System

by

Leonard Newton Foner

The following people served as readers for this thesis:

Reader

Peter G. Neumann
Principal Scientist
Computer Science Lab
SRI International

Reader

Deborah Hurley
Director, Harvard Information Infrastructure Project
Kennedy School of Government
Harvard University

Reader

Henry Jenkins
Professor of Literature
Director, Film and Media Studies Program
MIT Literature Department

Acknowledgments

This work could never have happened without the support and assistance of many people.

First and foremost, I thank my advisor, Pattie Maes, for her invaluable advice and encouragement in the years we have worked together.

I also thank the rest of my committee—Peter Neumann, Deborah Hurley, and Henry Jenkins—for their attention and advice.

I am forever grateful to Lisa Kamm for her unflagging friendship and support, and for her invaluable legal and political acumen. I am also deeply indebted to Michele Evard for her friendship and encouragement, and for helping to pass on the oral tradition that is so much a part of a Media Lab dissertation.

A large number of people contributed in one way or another to the development of Yenta and its ancillary systems. I thank Brad Rhodes for his friendship, for important feedback on certain aspects of Yenta's design, and for his development of the Remembrance Agent, whose document comparison engine has been passed back and forth, rewritten, and rearranged innumerable times between us and among several of our UROPs, whom I also thank.

Barry Crabtree, of British Telecom, was enthusiastic about Yenta from the beginning, not only contributing to an early prototype, but also in arranging for gorgeous animations from simulations of Yenta's network behavior.

Undergraduates, as part of MIT's UROP program, contribute mightily to many research projects and help make MIT what it is. Daniel Barkalow and Aaron Ucko have spent untold hours doing first-rate work on Yenta's code. Without their help, Yenta may never have been finished. They have my highest commendation and my most heartfelt thanks. In addition, Sofya Raskhodnikova, Edward Kogan, Bayard Wenzel, Aditya Prabhakar, and Katie King have made important contributions to one part or another of Yenta. I thank also Abhay Saxena, Peter Davis, Brian Sniffen, and Pamela Mukerji. Tomoko Akiba created Yenta's wonderful surrealistic icons, and Maggie Oh made its logo. Ray Lee wrote an excellent original prototype for Yvette, and Ivan Nestlerode upgraded and polished it until it was ready for prime time.

I also thank the authors of SSLeay, SCM, autoconf, automake, and gcc, without which this project could not even have been contemplated.

Finally, I would like to thank the many people not already mentioned above who have reviewed copies of this manuscript and provided comments on it, including David Anderson, Judy Anderson, Marlena Erdos, and David Bridgham.

Table of Contents

Chapter 1: Introduction 15
1.1 The fundamental premise 15
1.2 What's ahead? 16
1.3 What are we protecting? 16
1.4 The right to privacy 19
1.5 The problems with centralized solutions 22
1.6 Advantages of a decentralized solution 24
1.7 A brief summary of this research 25
1.7.1 The architecture and its sample application 26
1.7.2 Evaluation 26
1.8 Summary 27

Chapter 2: System Architecture 29
2.1 Introduction 29
2.2 Application traits 30
2.3 Application traits we are not considering 31
2.4 Yenta—the sample application 32
2.5 The overall architecture 33
2.6 Determining one user's characteristics 33
2.7 Bootstrapping 34
2.8 Forming groups of users—clustering 35
2.8.1 Data structures used in finding referrals and clusters 35
2.8.2 Referrals and clustering 35
2.8.3 Privacy of the information exchanged 38
2.9 What exactly is a cluster? 39
2.10 Using the resulting clusters 41
2.10.1 One-to-one communication 41
2.10.2 Broadcasting to all agents in a cluster 41
2.10.3 Hiding identities 42
2.11 Reputations 43
2.12 Running multiple agents on one host 44
2.13 Evaluation hooks 46
2.14 Summary 48

Chapter 3: Privacy and Security 49
3.1 Introduction 49
3.2 The problem 49
3.2.1 The threat model: what attacks may we expect? 49
3.2.2 How private is private? 51
3.2.3 Security design desiderata 51
3.2.4 Problems not addressed 53

3.3 Cryptographic techniques 54
3.3.1 Symmetric encryption 54
3.3.2 Public-key encryption 54
3.3.3 Cryptographic hashes 55
3.3.4 Key distribution 55
3.4 Structure of the solutions 56
3.4.1 The nature of identity 56
3.4.2 Eavesdropping 57
3.4.3 Malicious agents 57
3.4.4 Protecting the distribution 57
3.5 Selected additional topics 59
3.6 Summary 60

Chapter 4: The Sample Application: Yenta 63
4.1 Introduction 63
4.2 Yenta's purpose 63
4.3 Sample scenarios 63
4.4 Affordances 64
4.4.1 User interface 64
4.4.2 Yenta runs forever 64
4.4.3 Handles 65
4.4.4 Determining user interests 65
4.4.5 Messaging 66
4.4.6 Introductions 67
4.4.7 Reputations 67
4.4.8 Bookmarks 67
4.4.9 News 67
4.4.10 Help 68
4.4.11 Configuration 68
4.4.12 Other operations 68
4.5 Politics 68
4.6 Implementation details 69
4.6.1 The C code 69
4.6.2 The Scheme code 70
4.6.3 Dumping 71
4.6.4 Architectures 71
4.7 Determining user interests 71
4.7.1 Producing word vectors 71
4.7.2 Clustering 72
4.8 Security considerations 73
4.8.1 Encrypting connections 73
4.8.2 Protecting persistent state 73
4.8.3 Random numbers 77
4.9 Summary 77

Chapter 5: Evaluation 85
5.1 Introduction 85
5.2 Simulation results. 86
5.3 Collecting data from Yenta. 87
5.4 What data is collected? 89
5.5 A sample of results. 90
5.5.1 Qualitative results 91
5.5.2 Quantitative results 92
5.6 Security 93
5.7 Risk analysis 96
5.7.1 Denial of service 97
5.7.2 Integrity and confidentiality—protocols 98
5.7.3 Integrity and confidentiality—spies. 99
5.7.4 Contagion. 99
5.7.5 Central servers 100
5.7.6 Nontechnical risks 101
5.8 Other applications of this architecture 101
5.9 Motivating adoption of the technology 104
5.10 Future work 106
5.10.1 Sociological study 106
5.10.2 Political evaluation. 106
5.11 Summary 106
Chapter 6: Related Work. 109
6.1 Introduction 109
6.2 Matchmakers 109
6.3 Decentralized systems 111
6.4 Political software and systems 112
6.5 Summary 114
Chapter 7: Conclusions 117
References. 119

List of Figures

Figure 1: Yentas talk to each other and to their users' web browsers 33
Figure 2: Clusters and overlaps 39
Figure 3: Degrees of anonymity 51
Figure 4: Showing the user how to submit an evaluation. 61
Figure 5: A typical evaluation. The small bars on the left of each source line are color-coded. 61
Figure 6: A sampling of interests. Real users tend to have many more than shown here. 79
Figure 7: Recent messages received by this Yenta, and options for dealing with them. 79
Figure 8: A typical message, and how to reply. 80
Figure 9: Replying to a message. 80
Figure 10: Manipulating attestations. 81
Figure 11: Recent news about this particular Yenta. 81
Figure 12: A sampling of the help. 82
Figure 13: Adjusting internal parameters, for those who demand knobs.. 82
Figure 14: Some infrequently-used operations. 83
Figure 15: If Yenta is manually shut down, this is the last page it shows.. 83
Figure 16: Some selected statistics from fielded Yentas.. 93
Figure 17: Simulation results. See text for details.. 107

CHAPTER 1 *Introduction*

1.1 The fundamental premise

Technology does not exist in a social vacuum. The design and patterns of use of any particular technological artifact have implications both for the direct users of the technology, and for society at large. Decisions made by technology designers and implementors thus have political implications that are often ignored. If these implications are not made a part of the design process, the resulting effects on society can be quite undesirable.

The research advanced here therefore begins with a political decision: It is almost always a greater social good to protect personal information against unauthorized disclosure than it is to allow such disclosure. This decision is expressly in conflict with those of many businesses and government entities. Starting from this premise, a multi-agent architecture was designed that uses both strong cryptography and decentralization to enable a broad class of Internet-based software applications to handle personal information in a way that is highly resistant to disclosure. Further, the design is robust in ways that can enable users to trust it more easily: They can trust it to keep private information private, and they can trust that no single entity can take the system away from them. Thus, by starting with the explicit political goal of encouraging well-placed user trust, the research described here not only makes its social choices clear, it also demonstrates certain technical advantages over more traditional approaches.

We discuss the political and technical background of this research, and explain what sorts of applications are enabled by the multi-agent architecture proposed. We then describe a representative example of this architecture---the Yenta matchmaking system. Yenta uses the coordinated interaction of large numbers of agents to form coalitions of users across the Internet who share common interests, and then enables both one-to-one and group conversations among them. It does so with a high degree of privacy, security, and robustness, without requiring its users to place unwarranted trust in any single point in the system.

The research advanced here attempts to break a false dichotomy, in which systems designers force their users to sacrifice some part of a fundamental right—their privacy—in order to gain some utility—the use of the application. We demonstrate that, for a broad class of applications, which we carefully describe, this dichotomy is

indeed *false*—that there is no reason for users to have to make such a decision, and no reason for systems designers to force it upon them.

If systems architects understand that there is not necessarily a dichotomy between privacy and functionality, then they will no longer state a *policy* decision—whether to ask users to give up a right—as a *technical* decision—one required by the nature of the technology. Casting decisions of corporate or government policy as technical decisions has confused public debate about a number of technologies. This work attempts to undo some of this confusion.

The research presented here is thus intended to serve as an exemplar. The techniques presented here, and the sample application which demonstrates them, are intended to serve as examples for other systems architects who design systems that must manipulate large quantities of personal information.

1.2 What's ahead?

In this chapter, we shall:

Section 1.3
- Describe which type of privacy we are most interested in protecting

Section 1.4
- Discuss the concept of privacy as a right, not a privilege

Section 1.5
- Show some of the technical, social, and political problems with centralized manipulation of personal information

Section 1.6
- Show some of the advantages of a decentralized solution

Section 1.7
- Discuss the components of the work presented here, specifically its *architecture*, the *sample application* of that architecture, the *implementation* of that application, and issues of *deployment and evaluation*

Section 1.7
- Briefly summarize the remaining chapters of this dissertation

Later chapters will:

Chapter 2
- Discuss the system architecture for the general case

Chapter 3
- Analyze user privacy and system security

Chapter 4
- Detail the sample application—the matchmaking system Yenta

Chapter 5
- Discuss the evaluation of the architecture and of Yenta

Chapter 6
- Examine some related work

Chapter 7
- Draw some general conclusions

1.3 What are we protecting?

Privacy means different things to different people, and can be invoked in many contexts. We define privacy here as *the protection of identifiable, personal information about a particular person from disclosure to third parties who are not the intended recipients of this information*. This sentence deserves explanation, and we shall explain it below. We shall also touch upon some related concepts, such as *trust* and *anonymity*, which are required in this explanation.

Protection

Protecting a piece of information means keeping it from being transmitted to certain parties. Which parties are not supposed to have the information is dependent upon the wishes of the information's owner. This process is transitive—if party A willingly transmits some information about itself to party B, but party B then transmits this information to some party C, which A did not wish to know it, then the information has not been protected. Such issues of transitivity thus lead to issues of *trust* (see below) and issues of *assignment of blame*—whether the fault is in A (who *trusted* B not to disclose the information, and had this trust violated) or in B (who *disclosed the information without authorization* to C), or in both, depends on our goal in asking the question.

Identifiability

Unlinkability

In many cases, *disclosure* of information is acceptable if the information cannot be traced to the individual about whom the information refers—we refer to this as *unlinkability.* This is obvious in, for example, the United States Census, which, ideally, asks a number of questions about every citizen in the country. These answers to these questions are often considered by those who answer them to be private information, but they are willing to answer them for two reasons: The collection of the information is deemed to have *utility* for the country as a whole, and the collectors of the information make assurances that the information will not be *identifiable*, meaning that it will not be possible to know which individual answered any given question in any particular way—the respondents are *anonymous*. Because the Census data is gathered in a *centralized* fashion, it leads to a *concentration of value* which makes trust an important issue: central concentrations of data are more subject to institutional abuse, and make more tempting targets for outsiders to compromise.

Particular person

Whether or not the information is about a *particular* person—someone how is identifiable and is linkable to the information—or is instead about an aggregate can make a large difference in its sensitivity to disclosure. Aggregate information is usually considered less sensitive—although cross-correlation between separate databases which talk about the same individuals can often be extremely effective at revealing individuals again in the data, and represent a serious threat to systems which depend for their security solely on aggregation of data [169].

Personal information

When we use the term *personal information*, we mean information that is known *by* some particular individual *about* himself, *or* which is known to some set of parties who that individual considers to be authorized to know it. If no one else knows this information yet, the individual is said to *control* this information, since its disclosure to anyone else is presumably, at this moment, completely up to the individual himself. We are *not* referring to the situation whereby party A knows something about party B that B does not know about himself. Such situations might arise, for example, in the context of medical data which is known to a physician but has not yet (or, perhaps is not ever) revealed to the patent. In this case, B cannot possibly protect this information from disclosure, for two reasons: B does not have it, and because the information is known by someone who may or may not be under A's control.

Disclosure

If personal information about someone is not *disclosed,* then it is known only to the originator of that information. In this case, the information is still private. One of the central problems addressed by this dissertation is how to *disclose* certain information so that it may be used in an application, while still giving the subject control over it.

Third parties

Many existing applications which handle personal information do so by surrendering it, in one way or another, to a third party. This work attempts to demonstrate that this is not always required. In many instances, there is no *need to know*—knowledge of this information by the third party will not benefit the person whom this information is about. We usually use the term *third party* to mean some other entity which does not have a compelling need to know.

Intended recipients

The *intended recipient* of some information is the party which the subject desires to have some piece of personal information. If the set of intended recipients is empty, then the information is *totally private*, and, barring involuntary disclosures such as search and seizure, the information will stay private. The work presented here concerns cases where, for whatever reason, the set of intended recipients is nonempty.

Trust

Whenever private information is surrendered to an intended recipient, the subject *trusts* the recipient, to one degree or another, not to disclose this information to third parties. (If the subject has no trust in the recipient at all, but discloses anyway, either the subject is acting against his own best interests, or the information was not actually private to begin with—in other words, if the information is *public* and it does not mat-

ter who knows it, then there is no issue of trust.) Trust can be misplaced. A robust solution in any system, social or technological, that handles private information generally specifies that trust be extended to as few entities, in as minimal a way as possible to each one. This minimizes the probability of disclosure and the degree of damage that can be done by disclosure due to a violation of the trust extended by the subject.

Anonymity and pseudonymity

In discussing *unlinkability of information*, such as that expected by respondents to the US Census, we mentioned that the respondents trust that they are *anonymous*. To be fully anonymous is to know that information about oneself cannot be associated with one's physical extension—the actual individual's body—or with any other anonymous individual—all anonymous individuals, to a first approximation, might as well be the same person. This also means that the individual's real-world personal reputation, and any identities in the virtual world (such as electronic mail identification), are similarly dissociated from the information. Full anonymity is not always possible, or desired, in all applications—for example, most participants in a MUD are pseudonymous [20][33][49][59][60][116]. This means that they possess one or more identities, which may be distinguished from other identities in the MUD (hence are not fully anonymous), but which may not be associated with the individual's true physical extension. The remailer operated at penet.fi.net [77], for example, also used pseudonyms. There are even works of fiction whose primary focus is the mapping between pseudonyms and so-called *true names* in a virtual environment [176].

Reputations

The reason why the distinction between anonymity, pseudonymity, and true names matters has to do with *reputations.* In a loose sense, one's reputation is some collection of personally-identifiable information that is associated, across long timespans, with one's identity, and is known to a possibly-large number of others. In the absence of any sort of pseudonymous or anonymous identities, such reputations are directly associated with one's physical extension. This provides some degree of *accountability for one's behavior*, and can be either an advantage or a disadvantage, depending on that behavior—those with good reputations in their community are generally afforded greater access to resources, be they social or physical capital, than those with poor reputations. Pseudonymous and anonymous identities provide a degree of decoupling between the actions of their owners and the public identity. Such decoupling can be invaluable in cases where one wishes to take an action that might land the physical extension in trouble. This decoupling has a cost: because a pseudonym, and, particularly, an anonym, is easier to throw away than one's real name or one's body, they are often afforded a lower degree of trust by others.

A legal definition

Another way to look at the question of what we are protecting is to examine legal definitions. For a US-centric perspective, consider this definition from Black's Law Dictionary [14]:

> Privacy, Right of:
>
> The right to be let alone; the right of a person to be free from unwanted publicity; and right to live without unwarranted interference by the public in matters with which the public is not necessarily concerned. Term "right of privacy" is generic term encompassing various rights recognized to be inherent in concept of ordered liberty, and such right prevents governmental interference in intimate personal relationships or activities, freedoms of individual to make fundamental choices involving himself, his family, and his relationship with others. *Industrial Foundation of the South v. Texas*

> *Indus. Acc. Bd., Tex.,* 540 S.W.2d 668, 679. The right of an individual (or corporation) to withhold himself and his property from public scrutiny, if he so chooses.
>
> It is said to exist only so far as its assertion is consistent with law or public policy and in a proper case equity will interfere, if there is no remedy at law, to prevent an injury threatened by the invasion of, or infringement upon, this right from motives of curiosity, gain, or malice. *Federal Trade Commission v. American Tobacco Co.,* 264 U.S. 298, 44 S.Ct. 336, 68 L.Ed. 696. While there is no right of privacy found in any specific guarantees of the Constitution, the Supreme Court has recognized that zones of privacy may be created by more specific constitutional guarantees and thereby impose limits on governmental power. *Paul v. Davis* 424 U.S. 693, 712, 96 S.Ct. 1155, 1166, 47 L.Ed.2d 405; *Whalen v. Roe,* 429 U.S. 589, 97 S.Ct. 869, 51 L.Ed.2d 64. See also Warren and Brandeis, *The Right to Privacy,* 4 Harv.L.Rev. 193.
>
> Tort actions for invasion of privacy fall into four general classes: *Appropriation,* consisting of appropriation, for the defendant's benefit or advantage, of the plaintiff's name or likeness. *Carlisle v. Fawcett Publications,* 201 Cal. App2d 733, 20 Cal. Rptr 405. *Intrusion* [. . .] *Public disclosure* of private facts, consisting of a cause of action in publicity, of a highly objectionable kind, given to private information about the plaintiff, even though it is true and no action would lie for defamation. *Melvin v. Reid* 112 Cal. App. 285, 297 P. 91. [. . .] *False light in the public eye* [. . .]

1.4 The right to privacy

Constitutional arguments

Why is personal privacy worth protecting? Is it a right, which cannot be taken away, or a privilege, to be granted or rescinded based on governmental authority?

In the United States, there is substantial legal basis that personal privacy is considered a right, not a privilege. Consider the Fourth Amendment to the US Constitution, which reads:

> The right of the people to be secure in their persons, houses, papers, and effects, against unreasonable searches and seizures, shall not be violated, and no Warrants shall issue, but upon probable cause, supported by Oath or affirmation, and particularly describing the place to be searched, and the persons or things to be seized.

While this passage is the most obvious such instance in the Bill of Rights, it does not explicitly proclaim that privacy itself is a right.

There are ample other examples from Constitutional law, however, which have extended the rights granted implicitly by passages such as the Fourth Amendment above. Supreme Court Justice Brandeis, for example, writing in the 1890's and later, virtually created the concept of a Constitutional right to privacy [180]. For example, consider this quote, from *Olmstead v. United States* [130], writing about the then-new technology of telephone wiretapping:

> The evil incident to invasion of the privacy of the telephone is far greater than that involved in tampering with the mails. Whenever a telephone line is tapped, the privacy of the persons at both ends of the line is invaded, and all conversations between them upon any subject, and although proper, confidential, and privileged, may be overheard. Moreover, the tapping of one man's telephone line involves the tapping of the telephone of every other

> person whom he may call, or who may call him. As a means of espionage, writs of assistance and general warrants are but puny instruments of tyranny and oppression when compared with wire tapping.

Later examples supporting this view include *Griswald v. Connecticut* [71], in which the Supreme Court struck down a Connecticut statue making it a crime to use or counsel anyone in the use of contraceptives; and *Roe v. Wade* [147], which specified that there is a Constitutionally-guaranteed right to a personal sphere of privacy, which may not be breached by government intervention.

Moral and functional arguments

But the laws of the United States are not the only basis upon which one may justify a right to privacy—for one thing, they are only valid in regions in which the United States government is sovereign. It is the author's contention that there is a *moral* right to privacy, even in the absence of law to that effect, and furthermore that, even in the absence of such a right, it is a *social good* that personal privacy exists and is protected—in other words, that personal privacy has a *functional benefit*. In other words, even if one were to state that there is no legal or moral reason to be supportive of personal privacy, society functions in a more productive manner if its members are assured that personal privacy can exist. For example, there are spheres of privacy surrounding doctor/patient and attorney/client information which are viewed as so important that they are codified into the legal system of many countries. Without such assurances of confidentiality, certain information might not be exchanged, which would lead to an impairment of the utility of the consultation.

One might also argue that the *fear of surveillanc*e is itself destructive, and that privacy is a requirement for many sorts of social relations. For example, consider Fried [64]:

> Privacy is not just one possible means among others to insure some other value, but . . . it is necessarily related to ends and relations of the most fundamental sort: respect, love, friendship and trust. Privacy is not merely a good technique for furthering these fundamental relations; rather without privacy they are simply inconceivable.

For the purposes of this work, we shall take such moral and social-good assertions as *axioms*, e.g., not requiring further justification.

Implications for systems architects

Those who design systems which handle personal information therefore have a special duty: They must not design systems which unnecessarily require, induce, persuade, or coerce individuals into giving up personal privacy in order to avail themselves of the benefits of the system being designed. In other words, system architects have a moral, ethical, and perhaps even—in certain European countries, which have stronger data privacy laws than the US—legal obligations to design such systems from a standpoint that is protective of individual privacy when it is possible to do so.

There may be strong motives *not* to design systems in such a fashion that they are protective of personal privacy. We shall investigate some of the motives, with examples, in the next section, but overall themes include:

See Section 1.5.

- It is often conceptually far simpler to design a system which centralizes information, yet such systems are often easily compromiseable, either through accident, malice, or subpoena.
- The architects of many systems often have an incentive to violate users' privacy, often on a large scale. The business models of many commercial entities, especially in the United States, depend on the collection of personal information in order to obtain marketing or demographic data, and many entities, such as credit bureaus, exist solely to disseminate this information to third parties. The European Union

has data-protection laws forbidding this [47].

- Government intervention may dictate that users' privacy be compromised on a large scale. CALEA [21] is a single, well-known example; it requires that US telephone switch manufacturers make their switches so-called *tap-ready.*

Hiding policy decisions under a veil of technological necessity

In many instances, the underlying motives which lead to a system design that is likely to compromise users' privacy are hidden from view. Instead of being clearly articulated as decisions of policy, they are presented as requirements of the particular technological implementation of the system. For example, consider most Intelligent Transportation Systems [18], such as automated tollbooths which collect fees for use of roads. These systems mount a transponder in the car, and a similar unit in the tollbooth. It is possible, using essentially the same hardware on both the cars and in the tollbooths, to either have a *cash-based* system or a *credit-based* system. A cash-based system works like Metrocards in many subways—users fill up the card with cash (in this case, cryptographically-based electronic cash in the memory of the car's transponder), and tollbooths instruct the card to debit itself, possibly using a cryptographic protocol to ensure that neither the tollbooth nor the car can easily cheat. A transaction-based system, on the other hand, assigns a unique identifier to each car, linked to a driver's name and address, and the car's transponder then sends this ID to the tollbooth. Bills are sent to the user's home at the end of the month.

An example from the Intelligent Transportation System infrastructure

In other words, a cash-based system works like real, physical cash, and can be easily anonymous—users simply go somewhere to fill up their transponders, and do not need to identify themselves if they hand over physical cash as their part of the transaction. Even if they use a telephone link and a credit card to refill their transponders at home, a particular user is not *necessarily* linked to a particular transponder if the cryptography is done right. And even if there is such a linkage between users and transponders, there is no need for the system as a whole to know *where* any particular transponder has been—once the tollbooth decides to clear the car, there is no reason for any part of the system to remember that fact. On the other hand, a credit-based system works like a credit card—each tollbooth must report back to some central authority that a particular transponder went through it, and it is extremely likely that *which* tollbooth made this report will be recorded as well.

Cash vs credit

Both cash- and credit-based systems can use the same hardware at both the car and the tollbooth; the difference is simply one of software. In fact, the cash-based system is simpler, because each tollbooth need not communicate in real-time with a central database somewhere. (Tollbooths in either system must have a way of either detaining cars with empty or missing transponders, or logging license plates for later enforcement, but the latter need not require a real-time connection for the tollbooth to function.) Furthermore, a cash-based system obviates the need for printing and mailing bills, processing collections, and so forth.

Same hardware either way; cash is actually simpler

Yet it is almost invariably the case that requests for proposals, issued when such systems are in the preliminary planning stages, simply *assume* a credit-based system, and often *disallow* proposals which can enable a cash-based system. This means that such systems, from the very beginning, are implicitly designed to enable *tracking the movements of all drivers who use them,* since, after all, each tollbooth must remember this information for billing purposes. Furthermore, drivers are likely to demand itemized bills, so they can verify the accuracy of the data. (After all, it is no longer the case that they need worry only about the contents of their local transponder—they must worry about the central database, too.) Yet such a system can easily be used, either by someone with access to the bill mailed to an individual, or via subpoena or compromise at the central database, to stalk someone or to misuse knowledge about where the individual has been, and when. Large-scale data mining of such systems can infringe on people's freedom of assembly, by making particular driving patterns

ITS RFP's implicitly assume that drivers should be tracked

inherently suspicious—imagine the case whereby anyone taking an uncommon exit on a particular day and time is implicitly assumed to have been going to the nearby political rally. And even the *lack* of a record of a particular transit has already been used in court proceedings [18].

ITS RFP's are setting policy, not responding to technological necessity

The aim of the work presented in this dissertation is the demonstration that many, if not most, of these systems can be technically realized in forms that are as protective of users' individual privacy as one might wish. Therefore, designers of systems who fail to ensure their users' privacy are making a policy decision, not a technical one: they have decided that their users are not entitled to as much personal privacy as is possible to provide, and are implementing this decision by virtue of the architecture of the system.

Unnecessary polarization of the terms of the debate

While it is the author's contention that most such decisions are, at best, misguided, and at worst unethical, the fact that they are often disguised as purely technical issues polarizes the debate unnecessarily and is not a social good. If some system, whose capabilities would improve the lives of its users, is falsely presented as necessarily requiring them to give up some part of a fundamental right in order to be used, then debate about whether or not to implement or use the system is likewise directed into a false dichotomy. By allowing debate to be thus polarized, and by requiring users to trade off capabilities against rights, it is the author's contention that the designers and implementors of such a system are engaging in unethical behavior.

Legitimate reasons against absolute personal privacy

There may be many legitimate reasons why absolute privacy a system's users is undesirable. It is not the aim of this work to assert that there are no circumstances under which personal privacy may be violated; indeed, the moral and legal framework of the vast majority of countries presupposes that there must be a balancing between the interests of the individual in complete personal privacy, and those of the state or sovereign state in revealing certain information about an individual to third parties.

This work aims to decouple technical necessity from decisions of policy

However, we should be clear about the nature of this balancing. It should be dictated by a decision-making process which is one of policy. In other words, *what is the desired outcome?* It should not instead be falsely driven by assertions about *what the technology forces us to do.* The aim of this research is to decouple these two issues, for a broad class of potential applications, and to demonstrate by example that technological issues need not force our hand when it comes to policy issues. Such a demonstration by example, it is hoped, will also make clearer the ethical implications of designing a system which is insufficiently protective of the personal privacy of its users.

1.5 The problems with centralized solutions

It is often the case that applications which must handle information from many sources choose a centralized system architecture to accomplish the computation. Using a single, central accumulation point for information can have a number of advantages for the developer:

Why centralized solutions are handy

- It is easy to know where the information is
- Many algorithms are easy to express when one may trivially map over all the data in a single operation
- There is no problem of coordination of resources—all clients simply know where the central server is, and go there

Unfortunately, such a centralized organization has two important limitations, namely *reliability* and *trust*. Reliability is an issue in almost any system, regardless of the kind of information it handles, whereas trust is more of a serious concern in systems which must handle confidential information.

Reliability

A single, central point also implies a single point of failure. If the central point goes down, so does the entire system. Further, central points can suffer *overload*, which means that all clients experience slowdown at best, or failure at worst. And in systems where, for example, answering any query involves mapping over all or most of the database in a linear fashion, increasing the number of clients tends to cause load on the server to grow as $O(n^2)$.

Because of issues like this, actual large systems, be they software, business models, or political organizations, are often divided into a hierarchical arrangement, where substantial processing is done at nodes far from any center—if there even *is* a center to the entire system. For example, while typical banks are highly centralized, single entities—there is one master database of the value of each account-holder's assets—there is not a single central bank for the entire world. Similarly, the Internet gets a great deal of its robustness from its lack of centralization—for example, there is not a single, central packet router somewhere that routes all packets in the entire network.

Trust

Of greater importance for this work, however, is the issue of trust. We use the definition of *trust* advanced in Section 1.3, namely, trust that private information will not be disclosed.

It is here that centralized systems are at their most vulnerable. By definition, they require that the subject of the information surrender it to an entity not under the subject's direct control. The recipient of this information often makes a *promise* not to disclose this information to unauthorized parties, but this promise is rarely completely trustworthy. A simple taxonomy of ways in which the subject's trust in the recipient might be misplaced includes:

How might trust be violated?

- *Deception by the recipient*. It is often the case that the recipient of the information is simply dishonest about the uses to which the information will be put.
- *Mission creep*. Information is often collected for one purpose, but then used later for another, unforeseen purpose. In many instances, there is no notification to the original subjects that such repurposing has taken place, nor methods for the subjects to refuse such repurposing. For example, the US Postal Service sells address information to direct marketers and other junk-mailers—it gets this information when people file change-of-address forms, and it neither mentions this on the form, nor provides any mechanism for users to opt out. Often, the organization itself fails to realize the extent of such creep, since it may take place slowly, or only in combination with other, seemingly-separate data-collection efforts that do not lead to creep except when combined. Indeed, the US Federal Privacy Act of 1974 [175] recognizes that such mission creep can and does take place, and explicitly forbids the US government from using information collected for one purpose from being used for a different purpose—how the USPO is allowed to sell change-of-address orders to advertisers is thus an interesting question. Note, of course, that this Act only forbids the government from doing this—private corporations and individuals are not so enjoined.
- *Accidental disclosure*. Accidents happen all the time. Paper that should have been shredded is thrown away unshredded, where it is then extracted from the trash and read. Laptops are sold at auction with private information still on their disks. Computers get stolen. In one famous case in March 1998, it was revealed that GTE had inadvertently disclosed at least 50,000 unlisted telephone numbers in the southern California area—an area in which half of all subscribers pay to have unlisted numbers. The disclosure occurred in over 9000 phonebooks leased to telemarketing firms, and GTE then attempted to conceal the mistake from its customers while it attempted to retrieve the books. The California Public Utilities Commission had the authority to fine GTE $20,000 per name disclosed, an enormous, $1B penalty that was not actually imposed [9]. In March of 1999, AT&T accidentally disclosed 1800 email addresses to each other as part of an unsolicited electronic commercial mailing; Nissan did likewise with 24,000 [26].

- *Disclosure by malicious intent*. Information can be stolen from those authorized to have it by those intent on disseminating it elsewhere. Examples from popular media reports include, for example, IRS employees poking through the files of famous people, and occasionally making the information public outside of the IRS [173]. *Crackers*, who break into others' computer systems, may also reveal information that the recipient tried to keep private. There is often significant commercial value in the deliberate disclosure of other companies' data; industrial espionage and related activities can involve determined, well-funded, skilled adversaries whose intent is to compromise corporate secrets—perhaps to do some stock manipulation or trading based on this—or to reveal information about executives which may be deemed damaging enough to be used for blackmail or to force a resignation. Intelligence agencies may extract information in a variety of means, and entities which fail to exercise due diligence in strongly encrypting information—or which are prevented from using strong-enough encryption by rule of law—may have information disclosed while it is being transmitted or stored.
- *Subpoenas*. Even though an entity may take extravagant care to protect information in its possession, it may still be legally required to surrender this information via a subpoena. For example, Federal Express receives several hundred subpoenas *a day* for its shipping records [178]—an unfortunate situation which is not generally advertised to their customers. This leads to a very powerful general principle: *If you don't want to be subpoenaed for something, don't collect it in the first place.* Many corporations have growing concerns about the archiving of electronic mail, for example, and are increasingly adopting policies dictating its deletion after a certain interval. The Microsoft antitrust action conducted by the US Department of Justice, for example, entered a great many electronic mail messages into evidence in late 1998, and these are serving as excellent examples of when too much institutional memory can be a danger to the institution.

This is hardly a complete list, and many more citations could be provided to demonstrate that these sorts of things happen all the time. The point here is not a complete itemization of all possible privacy violations—such a list would be immense, and far beyond the scope of this work—but simply to demonstrate that the issue of trusting third parties with private information can be fraught with peril.

Is this software, or a business model?

Note that the discussion above is not limited to *software* systems. Replace *algorithm* with *business practice*, *client* with *customer*, and *central server* with *vendor*, and you have the system architecture of most customer/vendor arrangements. However, we shall not further investigate these structural similarities, except to point out that business models themselves often have a profound impact on the architecture of an application.

1.6 Advantages of a decentralized solution

Decentralized solutions can assist with both reliability and trust. Let us briefly examine reliability, and consider a system which does *not* contain a single, central, physical point whose destruction results in the destruction of the system. By definition, therefore, a single, physical point of failure cannot destroy this system. This says nothing about the system's ability to survive either multiple points of failure, nor its ability to survive a single *architectural* failure (which may have been replicated into every part of the resulting system), but it does tend to imply that particular, common failure modes of single physical objects—theft, fire, breakdown, accidents—are much less likely to lead to failure of the system as a whole. This is nothing new; it is simply good engineering common sense.

The issue of trust takes more examination. If we can build a system in which personal data is distributed, and in which, therefore, no single point in the system possesses *all* of the personal data being handled, then we limit the amount of damage—disclosure—that can be accomplished by any single entity, which presumably cannot con-

trol all elements of the system simultaneously. Systems which are physically distributed, for example, multiply the work factor required to accomplish a physical compromise of their security by the number of distinct locations involved. Similarly, systems which distribute their data across multiple administrative boundaries multiply the work factor required by an adversary to compromise all of the data stored. In the extreme case, for example, a system which distributes data across multiple sovereigns (e.g., governments) can help ensure that no single subpoena, no matter how broad, can compromise all data—instead, multiple governments must collude to gain lawful access to the data.

Cypherpunk remailer chains

Cypherpunks remailer chains [10][23][66] are example of using multiple sovereigns. A remailer chain operates by encrypting a message to its *final* recipient, but then handing it off to a series of intermediate nodes, ideally requiring transmission across multiple country boundaries. In one common implementation, each hop's address is only decodeable by the hop immediately before it, so it is not possible to determine, either before or after the fact, the chain of hops that the message went through. Properly implemented, no single government could thereby compromise the privacy of even a single message in the system, because not all hops would be within the zone of authority of any single government.

Costs of a decentralized solution

Of course, as applied to the applications we examine in this dissertation, the advantages of a decentralized solution do not come for free. They require *pushing intelligence to the leaves*—in other words, that the users whose information we are trying to protect have access to their own computers, under their own control. Decentralized systems are also somewhat more technically complicated than centralized solutions, particularly when it comes to *coordination* of multiple entities—for example, how are the entities supposed to *find each other* in the first place? And such solutions may not work for *all* applications formerly handled by centralized solutions, but only for those that share particular characteristics. We will investigate each of these issues in later chapters.

1.7 A brief summary of this research

The purpose of the work in this dissertation is to demonstrate that, for a class of similar applications, useful work that requires knowledge of others' private information may nevertheless be accomplished without requiring any trust in a central point, and without requiring very much trust in any single point of the system. In short, such a system is *robust* against violations of trust, unlike most centralized systems.

The work is therefore divided into several aspects, which will be discussed more fully in the chapters that follow, and which are summarized in this section:

- An *architecture* which specifies the general class of applications for which we are proposing a solution—what characteristics are common to those applications which we claim to assist? This architecture also includes our threat model—what types of attacks against user privacy we expect, which of those attacks we propose to address, and how we will address them. — Chapter 2, Chapter 3
- A *sample implementation* of this architecture—the matchmaking system Yenta. — Chapter 4
- *Evaluation* of the sample application as deployed, an analysis of the risks that remain in the design and implementation, and some speculations on how certain other applications could be implemented using the architecture we describe. — Chapter 5
- An examination of *related work*, both with regard to privacy protection via architecture, and the sample application's domain of matchmaking. — Chapter 6

1.7.1 The architecture and its sample application

We present a general architecture for a broad class of applications. The architecture is designed to avoid centralizing information in any particular place, while allowing multiple agents to collaborate using information that each of them possesses. This collaboration is designed to form groups of agents whose users all share some set of characteristics. The architecture we describe is particularly useful for protecting personal information from unauthorized disclosure, but it also has advantages in terms of robustness and avoidance of single points of physical failure. In the description below, the architecture and the sample application described in this dissertation—Yenta—are described together.

Such an architecture assumes several traits shared by applications which make use of it, of which the most important are the existence of a peer application for each user who wishes to participate, running on the user's own workstation; the availability of a network; the availability of good cryptography; and a similarity metric which can be used to compare some characteristic of users to each other and which enables a partial ordering of similarity. The architecture derives much of its strength from its completely decentralized nature—no part of it need reside on a central server. Users are pseudonymous by default, and agents are assumed to be long-lasting, with permanent state that survives crashes and shutdowns. Individual agents participate in a hill-climbing, word-of-mouth exchange, in which they exchange messages between pairs of themselves—with no central server participating in such exchanges. Agents which find themselves to be closely matched form clusters of similar other agents. An agent which is not well-matched to a peer can ask the peer for a referral to some other agent which is a better match, hence using word-of-mouth, based on the above partial ordering of similarities, to aid in the search for a compatible group of other agents.

Once clusters have been formed, agents may send messages into the clusters, communicating either one-to-one or one-to-many. Yenta uses this capability to enable users to have both private and public conversations with each other. Particularly close matches can cause one of the participating agents to suggest that the two users be introduced, even if the users have not previously exchanged messages—this helps those who never send public messages to participate.

We carefully discuss the threat model facing the architecture and the sample application, discussing which attacks are expected and the measures taken to defend against them. We also discuss what sorts of attacks are considered outside the scope of this research and for which we offer no solution. Strong cryptography is used in many places in the design, both to enable confidentiality and authenticity of communications, and as the infrastructure for a system designed to enable persistent personal reputations. Because public evaluation can make systems significantly more robust and more secure, a separate system, named Yvette, was created to make it easier for multiple programmers to publicly evaluate Yenta's implementation; Yvette is not specialized to Yenta and may be used to evaluate any system whose source code is public.

1.7.2 Evaluation

The architecture and the sample application have been evaluated in several ways, including via simulation and via a pilot deployment to real users. The qualitative and quantitative results obtained demonstrate that the system performs well and meets its design goals. In addition, several other applications which might make use of the underlying architecture are possible and speculations on how they might be implemented are briefly described. We also perform a risk analysis of Yenta and describe potential security risks, including some which are explicitly outside of our threat model.

Finally, we describe related work, which includes other types of matchmaking systems, other decentralized systems, and other systems and software that have been designed for explicitly political purposes. We then draw some general conclusions.

1.8 Summary

This chapter has presented the social and political motivations for this work, namely the protection of certain civil liberties, such as privacy, by starting with such motivations and then designing technology that can help. We have described what personal privacy and its protection means, demonstrated some of the social, political, and technical problems with centralized solutions, and touched upon some of the advantages of decentralized solutions. We have then summarized, very briefly, the work that will be presented in later chapters.

CHAPTER 2 System Architecture

2.1 Introduction

In this chapter, we present a general architecture for a broad class of applications. As discussed in Chapter 1, the architecture is designed to avoid centralizing information in any particular place, while allowing programs run by multiple users to collaborate by using information that each of them possesses. Such an architecture is particularly useful for protecting personal information from unauthorized disclosure, but it also has some advantages in terms of robustness various types of failure, including single points of physical failure.

This chapter will describe the architecture by answering the following questions:

- The traits shared by the applications we are considering — Section 2.2
- The problems are we *not* addressing in the space of possible applications — Section 2.3
- For concreteness, our sample application — Section 2.4
- The overall architecture proposed — Section 2.5
- Determining one user's *characteristics* — Section 2.6
- Bootstrapping — Section 2.7
- Forming groups of agents, including: — Section 2.8
  - Data structures used in clustering — Section 2.8.1
  - Getting referrals — Section 2.8.2
  - Privacy of the information exchanged — Section 2.8.3
- Further clarification on the exact nature of a cluster — Section 2.9
- Some uses for the resulting groups — Section 2.10
- Reputation systems — Section 2.11
- Running more than one copy of the application on a single host — Section 2.12
- Hooks for collecting evaluation data — Section 2.13

As discussed in Chapter 1, we obtain a large amount of our privacy and security protection from a *decentralized architecture*; that architecture is discussed in this chapter. We obtain other elements of protection from the techniques and principles advanced in Chapter 3; that chapter is heavily dependent upon this one.

Some technical privacy issues are explained in this chapter

In a few sections of this chapter, we delve into particular aspects of privacy and security in advance of Chapter 3's coverage. We do so because certain strategies for protecting user privacy are more easily explained near the description of some architectural feature than they are in a separate chapter.

Several issues are deferred

This architectural description defers several topics to later chapters. Some of the design decisions made here will be clearer when the entire picture has been presented. In particular, later chapters will specify:

Chapter 3
- How the privacy and security of the architecture really work

Chapter 4
- Details of how the sample application, Yenta, makes use of this architecture

Chapter 5
- How to evaluate how the system as a whole is performing

Chapter 5
- Other applications besides the sample application

2.2 Application traits

In the discussion that follows, we take *user* to be some individual person, *application* to be some particular user task which is implemented by running a program, and *system* to be a set of interconnected users, all running copies of some piece of code that implements the application. A familiar example of such a definition would be the *Internet mail system*, which consists of users all running applications (mail readers) which all do the same task, even though the applications themselves are not all identical—they run on different computers, come from different vendors, and have a different set of features which they implement. Note that the Internet mail system does not quite fit the definition given below of the applications we support; it serves only to make clear what we mean by *user, application,* and *system.*

Systems, applications, users, instances, and agents

For clarity, let us distinguish between the concepts of an *application* and an *instance of an application*. The *application* itself is the body of code that users may run; it is the same for all users who run the same version. The *instance* of that application is the individual copy that any given user is running on some machine, and includes whatever personalized state may exist for the user. In the discussion that follows, we refer to an individual instance of some running application as an *agent.* (Some examples and definitions of agents may be found in [16][27][30][31][45][46][59][60][88][98][101][106][112][113][114][143][159][160][162][163][164]—and many which are not listed there are mentioned at appropriate points elsewhere in this dissertation). We define an *agent* here to be a semiautonomous piece of software running on a particular computer, which may be personalized and has long-term state. We do *not* consider anthropomorphism or the ability to move the thread of control to another machine (e.g., process migration) to be a part of the definition we use here. The application is implemented by users running a distributed system of agents.

Let us turn to the traits which are shared by all the applications we are considering. Later sections will justify some of the assumptions and limitations.

- More than one user exists in the system. If there is only one user running the application, then we do not consider it a system.
- The users, and the agents they run, are all *peers* of each other. There is no distinguished user or agent, and no pre-established hierarchy.
- The application requires that some of its users wish to interact with some of the other users, by sharing some information between them.
- Not every user, nor his or her agent, need know about every other user or agent, nor does any user or agent require complete information about all other users or agents.
- It is appropriate to group users, on the basis of some attribute, into *clusters* which all share, to some extent, that attribute. Any given user might be in more than one cluster simultaneously, depending on the user's attributes.

- It is possible to form a *partial order* among user characteristics, such that we can say that some characteristic of user A is more like user B than user C.
- It is likely that at least some of the information in the system should be protected from disclosure to others, either inside the system or outside of it.
- Each of the users of the system can run their own copy of the application, on some computer at least nominally under their own control.
- The users are connected via a high-availability network, such as the Internet.

If there is no way to compare user characteristics, and no way to group users into even approximate clusters based on similarity of those characteristics, than many of the assumptions of our architectural model are violated. In particular, the architecture assumes that it can *climb a gradient* in order to form clusters (see Section 2.8), and that many operations are restricted to *users in a particular cluster.* If these are not true, then the architecture may not work very well. (Whether it works well enough even if some assumptions are violated is dependent upon exactly what the application is; we shall not further investigate what the properties of such an application might be.)

Because we are assuming that there exists information in the system that should be protected against others, and because of the arguments advanced in Chapter 1, particularly in Section 1.5, about the problems of *trust* when it comes to centralized systems, we assume that users must have the ability to do local processing of information they consider to be confidential. This requires that users have access to a computer that can run the application, and which they may be reasonably assured is under their administrative control, not that of some third party. Systems in which users must do computation in environments they do not control are explicitly not addressed by this work.

The applications we are considering are based around the controlled sharing of information between users. To this end, we assume that there is some way for the users' agents to actually *communicate* with each other, such that we define the set of agents as a *system.* For simplicity of discussion, we assume that this requires a network linking all agents in close to real time, e.g., the Internet. Generalizations of the fundamental architecture can certainly be made for *store-and-forward* networks, such as is usually assumed for mail transfer systems, and systems in which users are only infrequently connected—such as home users who only occasionally dial up to talk to the network—but we shall not explicitly address those considerations here. Most of the architecture we present is still usable in such a system, albeit with much greater delays between transactions between agents. Such delays may make the applications inconvenient to use in practice, even if they are theoretically still functional.

2.3 Application traits we are not considering

It is clear that the criteria above do not apply to all possible applications. For example, if there is only one user running the application, then we do not consider it a *system* at all. And if no user needs any information from any other user, then again it is not a *system*, because all the individual copies of the application do not interact with each other, and are running standalone, in a disconnected configuration.

By the same token, we assume that, even though users must communicate with each other, we never have 1-to-*n* or *n*-to-*n* interactions, where *n* is the set of all users or all agents in the system. There are two reasons to disallow such scenarios:

- *Robustness*. Systems in which any entity, or all entities, must see every other entity in the system tend to become extremely fragile as the number of entities grows. One way to see this, in a distributed system, is to take as a given that some probability *p* that some single entity will be offline for some reason—such as crashes,

network disconnections, and so forth. We assume that there is no redundancy (all entities must be online and known), that failures are independent of each other, and that there are *n* entities in the system. This means that the chance that the system as a whole scales exponentially poorly with *n*. Clearly, such a system will almost never function if *n* is large and *p* is not very close to zero.

- *Security.* Implementing the system as a *non*-distributed, e.g., centralized, system, can help with performance—if the central node is up, then presumably all information about all entities is known at that time and may be used. However, this still has unfortunate implications for security, since we have now established a single point of failure at which all entities' information may be compromised. If the system is instead decentralized, but all entities must still know all other entities' information, then the number of points where *all* entities' privacy may be compromised has now risen to *n*, the number of entities in the system. The situation is now worse, not better. We shall have much more to say about the security implications of our assumptions in Chapter 3.

2.4 Yenta—the sample application

For concreteness, let us mention here the *sample application*—*Yenta*—that has been developed. Yenta was developed both to test the architecture, and to serve as advertisement and role model for the technique. (Recall, from Chapter 1, that the purpose here is to encourage other developers and systems architects to use these techniques to avoid depriving users of their privacy in those other applications.) We will give much more information about Yenta's operation in Chapter 4—this is only a very brief summary.

Yenta is a *matchmaking system*. Yenta is *not* necessarily a romantic matchmaker. Instead, it is designed to facilitate *serendipitous introductions* of people who may or may not know each other, and to support *group interaction* among users who share common interests. Two possible scenarios of Yenta's use are:

- *Inside a company.* Many organizations often have the problem that people who *should* know what each other are doing do not. This is commonly the problem when two people are working on a similar problem, but report to different managers. In this case, it may be that the common point in their reporting structure is sufficiently high in the hierarchy that it fails to allow either of the two individuals to know about each other's work. While one might hope that the two individuals might meet accidentally and happen to mention their work to each other, such an event is not assured. (Even if the two do meet, they may fail to mention their common interest—it is rare that people regale each other with a list of *everything* they are working on at the moment.) Yenta aims to help, by serving as an introducer for these two, based on this common interest.
- *Among people who have never met.* Here, the problem is one of attention and interactional bandwidth. Even if we assume, for instance, that people who share a similar interest happen to both be on the same mailing list or Usenet newsgroup, not everyone posts. Indeed, if everyone *did* post, traffic volume might be so high that keeping up with the discussion might prove impossible. Yenta aims to help introduce *lurkers*—those who rarely or never post—to others who share their interests, without forcing them to speak publicly, and without subjecting everyone to the resulting traffic.

Each user runs his or her own copy of Yenta. Each Yenta determines its user's interests by scanning his or her electronic mail and files—this is one of many reasons why Chapter 3's discussion of privacy and security is so important. Agents join *clusters* of others, whose users share one or more interests, and users may send messages to individuals in the cluster or to the entire cluster as a whole. Users are pseudonymous, and their identities are never revealed by Yenta itself. (If a user sends a message to another that explicitly states his or her identify, that is not Yenta's concern.) Because pseudonyms are the norm, Yenta also makes available a *reputation system* to aid in deter-

mining whether to accept an introduction to another user, to help provide some context in interpreting another user's messages, or to enable automatic rejection of messages from users whose reputations are not good enough.

2.5 The overall architecture

The overall system architecture is a *distributed, multi-agent system*. Each user runs his or her own copy of the application—an agent. The agent has access to persistent, storage on the user's computer, e.g., a filesystem. This filesystem is used to store state across crashes and shutdowns. It may also be used for other purposes—for example, in Yenta, it is used as the source of the user's interests. The agent is assumed to run for long periods of time—effectively indefinitely—rather than being started up and shut down soon thereafter. It is thus assumed to be available to the user, and the rest of the network, most or all of the time. All communications and on-disk storage are assumed to be encrypted; Chapter 3 has much more to say about this requirement.

Agents communicate with each other by opening connections to each other across the network (using TCP [135] except in certain unusual circumstances, as below). Since not all copies of the given application should be assumed to be the same version, agents should identify themselves early in any given communication by specifying their current version information, a list of protocols or operations handled, or both—this aids in interoperability, allowing newer agents to be backwards-compatible with older agents where feasible.

Each agent must also be able to communicate with its user. We assume, for simplicity, that the user possesses a web browser, and the agent speaks HTTP [12][52] to that browser. This greatly simplifies design of the application, since emitting HTTP is a much easier implementation challenge than the engineering that goes into the typical browser.

A diagram of the basic structure appears below.

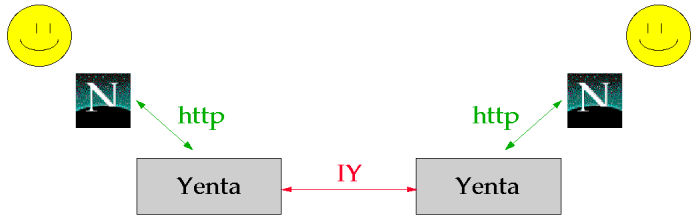

Figure 1: Yentas talk to each other and to their users' web browsers

2.6 Determining one user's characteristics

The architecture assumes that users have particular *characteristics* that make them suitable candidates for *clustering* into groups. Members of the group share at least one characteristic, to some degree, in common. How these characteristics are determined is in large part application-specific; we discuss the case for Yenta in Section 4.7.

We assume that these characteristics are *comparable* in some algorithmic fashion. We specified this in Section 2.2 when we said that we must have a partial order available in comparing one user's characteristics to another. In the case of Yenta (see Section 4.4.4), these characteristics are sets of weighted vectors of keywords, and the comparison is performed by dotting vectors together.

An example from Yenta

Any given user may have several characteristics. For example, in Yenta, any given user is presumed to have several interests at the same time. These characteristics are assumed to be sufficiently different from each other that our comparison function con-

siders them dissimilar from each other—if this were not the case, then at least two of these characteristics should be merged into a single characteristic.

2.7 Bootstrapping

When an agent is starting up for the very first time, it may not know, a priori, of any other agents for the application. In this case, it may use a bootstrapping phase in which it undergoes a discovery process that finds at least one other instance of the application. After this bootstrapping phase is accomplished, it need not be repeated.

This bootstrapping process can take many forms. Examples include:

- Broadcasting on the local network segment, for networks that support broadcasts
- Asking the user for any other machines known to be running the application
- Having existing agents periodically register their existence with a central server—the *bootserver*—and having newly-created agents ask this server for possibilities

Security of the bootserver

Yenta uses all three of these strategies. We shall have more to say about the security implications of this in Chapter 3; however, note for the moment that the only relevant aspect of this bootstrapping phase is that the agent find *any* other instance of itself with which to communicate. That instance need not share any of the user's characteristics. This makes design of the bootstrap server both simple and secure, since it need not maintain any identifiable user information, except the IP address at which some agent was found recently—for most applications, this is not a serious infringement upon user privacy. If the database is accidentally destroyed, it will be regenerated as running agents periodically register. The central server may also, of course, be specific to a particular organization if desired, rather than there being a single such server on the entire Internet.

Note that if the application being considered is so ubiquitously deployed that the chances are very high of another one of its agents existing on the local broadcast network segment, or of a new user already knowing of another agent, the central server becomes redundant.

Bootstrap broadcasts are very different from cluster broadcasts

Be aware that agent broadcasts, used in the sense we mean here for bootstrapping, are *not* the same sort of mechanism that we specify in Section 2.10, when we talk about communicating with a group of other agents. This is an important distinction:

- *Cluster broadcasts*, as described in Section 2.10, use *encrypted*, *point-to-point* transmission of messages, which are then recursively flooded to neighboring agents using the same mechanism. The flooding algorithm is designed to prevent loops by detecting graph cycles. Messages are transmitted via TCP [135].
- *Bootstrap broadcasts*, as described here, use *cleartext*, *broadcast-medium* transmission. On IP networks, this use is accomplished via UDP [134], since UDP supports broadcast, whereas TCP does not. Since we are not transmitting any personal information in a bootstrap broadcast—indeed, since the broadcasting agent may not *have* any yet—and since the message is intended for maximum reception, we do not encrypt its contents.

Broadcast responders must wait a random time before responding!

For broadcasting to work, all agents must be prepared to listen for, and respond to, bootstrap broadcasts. In general, both broadcast requests and replies should include information about the application—to enable multiple applications to share the same port—and its version—to enable backwards-compatibility with older applications. In addition, listeners on Ethernet-like [82] networks *must* implement *random delay* in their responses, so as to avoid a *packet storm* due to collisions on the wire caused by many agents responding at exactly the same time. Ethernet implementations are generally designed to incorporate *random exponential backoff*, such that collisions cause all transmitters to wait a random, exponentially-increasing amount of time before

each retransmission, but such packet storms can still last tens of seconds on a network segment with many responders. In the case of Yenta, for example, agents responding to a broadcast wait a random time, continuously and uniformly distributed between 0 and 2 seconds, before responding to any request. Since transmitting a packet takes between 10 and 100 microseconds, the chances of many responses colliding are negligible.

2.8 Forming groups of users—clustering

We now come to the core idea which makes our distributed system function, namely *how agents are supposed to find each other* and how they *organize into clusters*.

Any given agent starts knowing at least one other agent, via the bootstrapping mechanisms described in Section 2.7 above. Agents then use *one-to-one* communication of their characteristics, and a *referral algorithm*, to find suitable clusters.

2.8.1 Data structures used in finding referrals and clusters

For concreteness, assume that we have two agents, named A and B, which each have a few characteristics associated with them, e.g., C_{A0}, C_{A1}, etc. Each of these characteristics describes something about the agent's user. Each agent also contains several other data structures:

- A *cluster cache*, *CC*, which contains, for each characteristic, the names of all other agents currently known by some particular agent as being in the same cluster for that characteristic. Thus, if agent A knows that its characteristic 1 is similar to characteristic 3 of agent B, then CC_A contains an entry linking C_{A1} to C_{B3}. There are two important limits to the storage consumed by such caches: the number of *local characteristics*, c_l, that any given agent is willing to remember about itself; and the number of *remote characteristics*, c_r, that this agent is willing to remember about other agents. The total size of *CC* is hence bounded by c_1 times c_r. In an implementation that wishes to save space, limiting c_r before limiting c_l makes the most sense, as this limits the total number of other agents that will be remembered by the local agent, while not limiting the total number of disparate characteristics belonging to the user that may be remembered by the local agent.
- A *rumor cache*, *RC*, which contains the names and other information, as described below, from the last *r* agents that this agent has communicated with. Implementations should bound this number, since otherwise any given agent will remember *all* of the agents it has ever encountered on the net and its storage consumption will grow monotonically. Reasonable values for bounds are application-specific; Yenta uses values of 20 to 100.
- A *pending-contact* list, *PC*, which is a priority-ordered list of other agents that have been discovered but which the local agent has not yet contacted.

The rumor cache contains more than just the names of other agents encountered on the network. It also contains some subset, perhaps complete, of the value of each characteristic corresponding to those agents. Exactly how much of each characteristic is stored is application-specific.

2.8.2 Referrals and clustering

Now that we have all this mechanism in place, performing referrals and clustering is relatively uncomplicated.

Comparing one agent with another

The process starts when some agent (call it A) has ascertained its user's characteristics, and has found at least one other agent (call it B) via bootstrapping. The two agents exchange characteristics. Agent A then performs a comparison of its local characteristics with those of agent B. Agent A builds an upper-triangular matrix describing the similarities between each of its local characteristics and those locally held by B. Then it finds the highest score(s)—e.g., closest similarity—between any given characteristic (say, C_{A1}) and B's characteristics. If there is no such value above a particular threshold, then the local characteristic under consideration does not match

any of B's characteristics, although some other local characteristic, e.g., C_{A2}, might match.

Note that this inter-agent similarity metric cannot, in general, assume that it knows about all or even most of the other agents on the network. Hence, algorithms which assume that they can take means or do standard deviations to compute whether this is a particularly good match do not have the data to make this determination. Instead, the application must either use fixed thresholds, or attempt to refine its criteria after seeing some number of other agents' characteristics—which implies that the comparison metric is nonmonotonic, e.g., that it may behave differently for different inputs based on its prior history. In the sample application—Yenta—a simple thresholding scheme is used.

When we are done comparing characteristics from A with characteristics from B, agent A may have found some acceptably close matches. Such matches are entered, one pair of characteristics at a time, in A's cluster cache. B is likewise doing a comparison of its characteristics with A and is entering items in its own cluster cache for its own use.

Comparisons are not symmetric

Since each agent is making its own determination of similarity, and since they may be running different versions of the application, or have different local data available—nothing specifies that an agent must transmit *all* of its information about a particular characteristic to any given other agent—they may reach different conclusions. In other words, A may decide that B shares some characteristic with A, whereas B may not decide that it shares any characteristics with A. This asymmetry is perfectly acceptable. In the case above, it means that A will enter B in its cluster cache for some characteristic, but B will not enter A in its cluster cache for any characteristic.

Getting referrals

Whether or not any matches were found that were good enough to justify entering them in a cluster cache, the next step is to acquire *referrals* to agents that might be better matches. In the example here, agent A asks agent B for the entire contents of its rumor cache, and runs the same sort of comparison on those contents that it did on agent B's own local characteristics—but with a more forgiving threshold for what constitutes a good match. For example, if the comparison metric were to return a value between 0 and 1, ranging from no match to perfect match, then the threshold used to determine whether to add some characteristic from B to A's cluster cache might be 0.9, while the threshold used to determine whether a rumor-cache match is good enough might be 0.7.

The purpose of using a more forgiving threshold is to allow A to find someone else who might be reasonable, even if they aren't a great choice. Agent A will then add the agent corresponding to each such match to its pending-contact list, and will contact them in turn.

Agent A, having now acquired some likely candidates, will execute the same algorithm it just used with B: It will see if any of the agents is suitable to be added to A's cluster cache, and will also find other candidates who might be worth contacting. If the pending-contact list is kept sorted by desirability—presumably, by sorting the pending agents to contact by the result of the comparison metric—then A is executing a *hill-climbing* algorithm to finding a good match. In other words, if we model a *landscape* in which the height of any given hill is its similarity to some characteristic of A's, and A's current set of candidates as some point on the hillside, A should attempt to always travel in the direction of maximum upward gradient, essentially climbing hills in this space until it reaches a maximum. Note that we are climbing a *different* landscape, composed of *different* hills, for each characteristic.

Hill-climbing versus local maxima

Hill-climbing algorithms can get stuck at local maxima which are not global maxima. In practice, this appears not to happen in our sample application, neither in simulation nor in actual use. To get stuck at a local maxima requires that the system act *thermodynamically cold*, in the sense of simulated annealing. Here the metaphor is one of energy—a marble rolling around in a potential well cannot escape this well unless it possesses enough energy to roll *uphill* past an adjacent peak. Similarly, one balanced on a hillside might roll into the valley, but cannot hope to reach an even higher hilltop unless it something gives it extra energy. Random additions of extra energy—which may eventually roll a marble out of a stuck state—are thus similar to heating a system, hence we can talk about the thermodynamic temperature of a system.

Real data appears to be noisy enough that local maxima which are not global maxima are not a problem—there is enough inaccuracy in the comparison function, and in the data it is applied to, that agents do not get stuck. Furthermore, in a real system, one might expect that agents are constantly joining (and perhaps leaving) clusters, which will also tend to disrupt many such local maxima—it only takes one new agent that is a little better matched to knock some agent off its local maximum.

It is entirely possible that one can generate disconnected islands of agents which do not know about each other, and there is no feasible way to completely eliminate this possibility if we assume—as we do explicitly in Section 2.2—both that there is no central point in the system that knows about all agents, and that no agent is required to know about all others. However, such islands are likely to be rare, for several reasons:

- The bootstrap server (see Section 2.7) tends to tell brand-new agents about many existing agents, all over the world, which tends to ensure a wide sample of starting agents.
- It only takes one bridge between two formerly-disconnected islands to inform a large numbers of agents about each others' existence. The referral algorithm tends to encourage this behavior, since many agents will spread the news.

Metrics must allow a partial order

Of course, for this to work at all, the comparison metric must make available a gradient, via a partial order, as specified in Section 2.2—this is why the comparison function must not be a simple, binary predicate. Exactly *how* this predicate works is application-specific, but it must return some scalar value that we can compare. Issues of thermodynamic noise also tend to avoid pathologies, such as partial orders that lead to cycles (A>B>C>A). It may be the case that some applications can suffer from this problem; but we have not observed it here, and determining the exact conditions under which such pathologies might occur is beyond the scope of this work.

If we do not have a comparison metric which allows hill-climbing, then the referral process degenerates to a process more resembling diffusion in a gas—each agent simply explores the space of other agents at random. Results will still be obtained in this scenario, but very slowly—the situation goes from something approximately $O(n)$ to $O(n^2)$. Another way to look at this is to imagine that each agent is walking around in some physical space: a gradient-driven process moves the agent $O(n)$ steps from the origin, where n is the number of iterations, whereas a random process moves the agent only $O(\sqrt{n})$ steps from the origin.

Cluster cache is not for third-party data

Note that agent A never adds some agent, say W, to its cluster cache on the basis of B's say-so. After all, B's idea of W's characteristics could be wrong for any number of reasons. For example:

- W's data might be out-of-date or otherwise stale.
- W might have deliberately omitted some data in its transmission to B, perhaps based on some aspect of B's network address or reputation (see Section 2.11).
- B's idea of W's data might not even truly belong to W at all—see Chapter 3 for why this might be so.

For all of these reasons, we use B's rumor cache information only to add potential candidates to A's pending-contact list. When A eventually contacts any given candidate, a good match will be added to A's cluster cache in the usual way.

Referrals are like human word-of-mouth

This procedure acts somewhat like human word of mouth. If Sally asks Joe, "What should I look for in a new stereo?" Joe may respond, "I have no idea, but Alyson was talking to me recently about stereos and may know better." In effect, this has put Alyson into Sally's pending-contact list (and, if Joe could quote something Alyson said that Sally found appropriate, perhaps into Sally's cluster cache as well). Sally now repeats the process with Alyson, essentially hill-climbing her way towards someone with the expertise to answer her question.

2.8.3 Privacy of the information exchanged

The description so far suffers from a number of unfortunate security problems. For instance, when agent A sends its characteristics to agent B, B knows everything that A sees fit to tell it—and also knows A's IP address, hence making backtracing the information to the actual *user* possibly very easy. Furthermore, B will propagate information about A to any third parties which may care to ask B for its rumor cache, and this will continue to be true until B decides to flush A's information from its rumor cache—which could be never, since when to flush this information is entirely at B's discretion.

We have two strategies for avoiding this outcome: *hiding the identity* corresponding to any given characteristic, and *mixing others' clusters into the local user's data.* In practice, we do both.

Hiding identities via random reforwarding and digital mixes

We can use several strategies to hide the identity corresponding to a given characteristic. Techniques related to *random reforwarding* and *digital mixes* are discussed more extensively in Section 3.4.3. They depend both on anonymity of individual agents and the ability to broadcast into groups of agents, using keys known only to a subset.

Plausible deniability via other agents' data

One way of establishing a user's *probable or possible innocence*—in the terminology of Section 3.2.2—without having to go to the extremes of Section 3.4.3 is by including other users' data with our own. To enable plausible deniability of characteristics, it suffices for an agent to *lie.* In addition to offering its own characteristics, the agent can offer some characteristics that are currently stored in its rumor cache. By definition, such characteristics are not only not those of the offering agent, but they do not even reflect any of its own characteristics accurately—if they did, they would be in the agent's *cluster* cache, not its *rumor* cache. The agent offering the characteristics certainly knows which ones came from its cluster cache—and thus reflect the characteristics of its user—and which came from the rumor cache—and thus do not. However, the agent receiving these characteristics has no way to know.

Depending on the size of its rumor cache, the deceitful agent could easily be able to offer, say, ten times as many characteristics as it really owns. Thus, the probability of any single characteristic offered by the agent actually reflecting some characteristic of its user would be only 10%. Assuming that an agent is willing to store arbitrarily many characteristics in its rumor cache—and is willing to subject it and all of its peers to an arbitrary amount of work—this percentage can be made arbitrarily low.

In order to know which characteristics actually belong to a given agent, an attacker would have to be a party to many exchanges, looking for those characteristics which are *always* offered—such characteristics presumably correspond to the real characteristics of the agent's user. This attack could only work if the agent of interest either offers only subsets of its rumor cache, or runs long enough to flush entries from its rumor cache. A local eavesdropper—one who can listen to all of the given agent's traffic—could not accomplish this, because we assume, as advanced in Chapter 3, that

all communications are routinely encrypted. Instead, the attacker would have to actually compromise many agents on the network, and each of those agents would have to interact with the target agent, for the attack to succeed. While this is possible, it violates our assumption in Section 3.2.1 that an attacker does not control an arbitrarily high proportion of all agents with which the target agent interacts.

2.9 What exactly is a cluster?

In the discussion above, we have used the term *cluster* as if it denotes a particular, well-defined group of agents, and as if all agents within the cluster agree on its membership. This is not in fact the case. Let us examine the meaning of a cluster more closely.

A cluster is not a simple transitive closure

Consider the point of view of a single agent A, which believes itself to be in a cluster of agents which share characteristic C. This cluster is composed of all other agents in A's cluster-cache for C. It is also composed of all of *their* cluster-cache entries for characteristic C, and so on. In other words, if we treat the existence of some agent B in some agent A's cluster cache as a unidirectional link from A to B, then A's cluster is the *transitive closure*, starting from A's cluster cache for C, of all agents which are reachable by traversing these links. The links are unidirectional, e.g., forming a digraph and not a graph, because membership in a cluster cache is not guaranteed symmetric—see Section 2.8 above.

If all agents shared exactly the same value for C, then this definition could be recursively enumerated by A, simply by walking this digraph, keeping track of which agents have been visited, in the manner of a mark-sweep garbage collector [90]. One might argue that A *shouldn't* walk this digraph—this would eventually result in A having to remember every agent in its cluster, which violates the architecture criteria in Section 2.2—but it would at least be theoretically possible.

Characteristics are likely to be unique

However, all agents presumably do *not* have exactly the same value for C. We assume that characteristics may be complicated entities, capable of taking on a large number of values. For example, in Yenta—see Chapter 4—characteristics are weighted vectors of keywords. In this application, the exact makeup and weighting of any vector is unlikely to be reproduced by any other agent.

An example from Yenta

Continuing our Yenta-based example, suppose that we have three agents, each with slightly different interests. Yenta X's user is interested in cats. Y's user is interested in both cats and dogs. Z's user is interested in dogs. A schematic of this situation appears in Figure 2 below, where ellipses represent—approximately—the set of agents each Yenta considers to be in its own cluster. Note that the cluster names, $C_{1\text{-}3}$, are for explanatory convenience only—as we stated immediately above, clusters have no overall name of their own, but are described only by the set of which agents consider themselves to have similar characteristics.

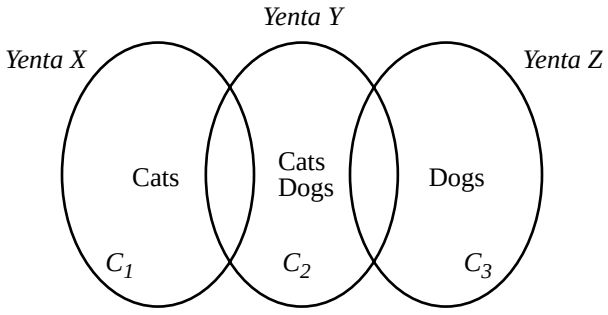

Figure 2: Clusters and overlaps

Assume, for the sake of discussion, that the metric which compares interests looks only at overlaps in words in the keyword vectors exchanged. This means that X and Y consider themselves to be in cluster C_1 (they are both interested in cats), and Y and Z consider themselves to be in cluster C_2 (they are both interested in dogs). However, should X and Z consider themselves to be in the same cluster?

The answer is no. X and Z are not both in C_1, C_2, or even some third cluster, given the interests expressed here. As far as we can tell from the comparison metric—which states that a shared interest must involve an overlap in keywords—X and Z are not interested in the same thing.

What is a gerrymandered cluster?

This means that X should *not* walk the digraph of all *other* agents' cluster-cache entries in order to compute which other agents are in its cluster—to do so would incorrectly cause X to believe that Z is in cluster C_1, when it most clearly is not.; Z's user has no interest in cats. We refer to such an outcome—in which X would believe that Z is in cluster C_1—to be a *gerrymandered* cluster. We use this term by analogy with its political use: a gerrymandered electoral district is one that has been stretched out of its natural shape—generally one with close to minimal circumference for its area—into one that unnaturally includes areas that seem better connected to different districts. Similarly, a gerrymandered cluster is one that unnaturally includes too many characteristics which, in reality, have nothing to do with each other. In effect, viewing interests as areas, such a cluster is stretched out in nonsensical ways.

Trusting other agents' judgments leads to gerrymandered clusters

Why would this happen? Because X, in recursively enumerating the members of cluster C_1, would be trusting the judgment of Y about what an interest really means. As far as Y is concerned, it is in a single cluster, C_2, which happens to specify interests which mention either cats or dogs. But this is not a view shared by either X or Z, whose interests are more restrictive.

No global ontology

No distinguished cluster names

Remember that nowhere have we stated that characteristics (in the general case) nor interests (in the case of Yenta) have *distinguished names* or some other attribute that would make them unambiguously identifiable as being the same, or different, across all agents in the system. We have provided no central authority to impose a consistent ontology on all agents in the system. Furthermore, for all agents to reach a consensus among themselves, we would have to provide some mechanism to permit, in the limit, propagating such a proposal to the entire system and making it consistent. We have provided no such mechanism. Instead, we provide only the assurance that there exists a metric which can compare *one agent's characteristics with another* and to reach a local, not a global, decision about similarity of characteristics.

Thus, one agent should not trust another about what a characteristic for a third agent really means, because one agent has no assurance that another shares its ontology. All such judgments must necessarily be local—meaning that, if X is to make a determination about whether Z shares some characteristic with it, it needs to examine Z's data directly. It cannot trust the judgment of some intermediate agent Y. This does *not* mean that X must communicate directly with Z to make this determination, however. As long as X may be assured that it receives a faithful copy of Z's data, no matter where this copy comes from, X may make the comparison. But it must make the comparison *itself*.

2.10 Using the resulting clusters

Once we have clustered agents based on characteristics shared by their users, what can we do with the resulting clusters? We shall investigate some uses of these clusters below. Applications which fit the criteria advanced in Section 2.2, but are substantially different from Yenta, may have additional uses for these clusters.

The basic operations we will investigate here concern:

- Communicating from one user to a single other user
- Broadcasting to all other users in a cluster
- Hiding the origin and destination of communications

By the end of this subsection, we shall also have derived the rationale and use for the basic components of any message transmitted—namely, a tuple consisting of the message itself, a unique-ID, and a cluster characteristic. Many ways of presenting such messages are possible; their real-time or close to real-time nature makes it reasonable to use an email-like user interface, or something akin to Zephyr instances [1][36].

2.10.1 One-to-one communication

In the simplest scenario, one agent simply transmits a message to some other agent, using the same sort of network connections as are used to swap characteristics. Whether or not the two agents are in the same cluster is irrelevant—once one of the agents has found the IP address of another, a connection may be opened. However, it is presumed that most such communications are between agents which believe each other to share characteristics—loosely, they are in the same cluster—because we presume that users who share characteristics have the most to say to each other.

2.10.2 Broadcasting to all agents in a cluster

.A more complicated scenario involves sending a message to all other agents in a cluster. In this case:

- The broadcasting protocol should be *efficient*, and must *terminate.*
- We must handle the case of *gerrymandered clusters*, as described in section Section 2.9.

Efficiency

Efficiency in the protocol means that no one agent should be required to do all the work of communicating with all other agents in its cluster. (Indeed, as shown in Section 2.9, it cannot even determine exactly what all the other agents in the cluster *are.*) Hence, the way we implement broadcasts is to use a *flooding* algorithm, familiar from the Usenet news system [83]. When an agent wants to send a message to all other agents in its cluster, it sends it to all other known agents in its cluster cache, with instructions that the message should be forwarded to all other agents in *their* cluster caches, and so on recursively.

Termination

If this was the entire protocol, it would fail to *terminate*, because the possibility exists that there will be cycles in the digraph describing which agents are in which other agents' cluster caches. A message sent into this graph would circulate endlessly. To avoid this, messages are tagged with a *unique identifier (UID)*, and every agent compares incoming broadcast messages with a cache of recently-seen UID's. If this message has been seen before, it is dropped immediately, and not propagated.

The UID cache in each agent must preserve incoming UID's long enough that there is a low probability that the message might still be circulating by the time it is timed out of the cache. This probability need not be zero, and cannot be: If we assume bounded storage in any given agent, but also assume that any agent may receive a message, crash, and then stay down an arbitrary length of time before coming back up and attempting to send the message, then we cannot set any particular timeout that is long enough. Instead, we must merely guarantee that the *effective gain* of the system—the number of messages emitted by any given agent, on average, for a single message

received—is low enough that messages are eventually damped out. If this is the case, then circulating messages will eventually vanish from the system, even though any given agent may occasionally see a duplicate message from some time far in the past. (Applications which cannot ever tolerate a duplicate message must arrange to maintain UID's forever, or must reject messages older than a certain age as part of their filtering algorithm.)

Avoiding gerrymandering

We now turn to the case of *gerrymandered clusters*. Consider the case of the three example Yentas described above in Section 2.9. Suppose that Yenta X wishes to broadcast to its cluster. Clearly, Y should receive such a broadcast, because the two Yentas share an interesting in cats. However, Z has no interest in such a message, nor would any other Yentas in C_3. This means that Z must have some way to know that it should drop the message—otherwise, messages intended for what X considers C_1 (and what Y considers C_2) would also propagate into C_3, and presumably far into clusters beyond as well.

To avoid this scenario, messages that are transmitted also include the *characteristic* which describes the cluster, from the point of view of the *original sender* of the message. It is very important that this is the *original* sender's characteristic—if this were not the case, then third-party recipients of the message (Z in our example) would again be heeding some intermediate party's idea of what a given cluster was about. Given that the characteristic is transmitted along with the message, each agent in the chain can evaluate whether the message still seems relevant to its own set of clusters. If the message is relevant to none, then it is dropped. (Note that it is possible that X's original characteristic might be deemed to match more than one cluster in some receiving agent; in that case, the message should be duplicated and broadcast into each cluster.)

In order to aid agents receiving one-to-one (non-broadcast) messages, and to make the protocol simpler by increasing commonality between the two cases, we also transmit the relevant characteristic along with the message even in the one-to-one case. We can only do this if the transmitting agent actually knows which cluster the recipient's agent is in; it may be the case that the user wishes to transmit a message to a particular agent irrespective of its cluster. In this case, no characteristic will be sent.

A complete message tuple

We have thus arrived at the complete set of tags that must accompany any given message between agents. A complete message thus consists of:

- The message itself.
- The message's UID.
- The characteristic associated with the cluster—required if a broadcast, suggested if one-to-one.

2.10.3 Hiding identities

Let us now consider the case in which it is important to hide the identify of the sending or receiving agent. We shall investigate this case in more detail in Chapter 3, but we should point out here that this capacity is important to make available. Without the ability to hide message originators and recipients, *traffic analysis* may be employed to guess information about the agents in the system.

For example, given the three-Yenta scenario in Section 2.9, suppose that we are an eavesdropper who can monitor communications between agents, even though we may not be able to decrypt them. If we know, though some mechanism, that Yenta X is interested in cats, and see substantial message traffic between X and Y, we can make a reasonable guess that Y is interested in cats as well.

The easiest way to defuse this threat is to send any message for a given agent in a cluster to *all* agents in the cluster—in other words, to broadcast it. Assuming that the connectivity of the cluster, and the characteristics of each agent in it, are suitable, we have an arbitrarily high probability that the target agent will receive at least one copy of the message. Obviously, if the message is also intended to be *private*, it must be encrypted using a key that only the recipient knows; we will address this more fully in Chapter 3. All agents which receive the broadcast attempt to decrypt it, but only the target agent possesses the correct private key; all other agents fail to decrypt the message and simply drop it. This is the general idea behind *Blacknet* [118], an idea suggested in the Cypherpunk community as a way to anonymously trade secrets, yet foil traffic analysis, by broadcasting any given message to the entire world via Usenet news, yet encrypt it only for its intended recipient(s).

This means that, in the general case, even one-to-one messages are broadcast. They are propagated, as part of foiling traffic analysis, by all agents which deem the message to be close enough to one of their existing clusters. Because actual message being propagated is encrypted, it may only be read by a subset—possibly singular—of the agents. This is clearly not as conservative of network resources as direct, point-to-point connections, but it is far safer if widespread eavesdropping and traffic analysis is considered to be a threat. If proper Mixmaster [10][23][66] dithering of the timing and size of transmissions is employed—by padding all messages to the same size, sending garbage messages when there is nothing to send, and sending messages either at totally random times or totally periodic times—it is possible that both sender and receiver could be *beyond suspicion*, as in the definition in Section 3.2.2.

We will address further aspects of this mechanism, including its behavior against active attackers and widespread traffic analysis, in Chapter 3.

2.11 Reputations

It is expected that this architecture will be used for applications which handle personal data. Much of the strength of the privacy-protecting features of the architecture (see Chapter 3) derives from the use of pseudonyms in place of real user identities.

Trolling and spoofing

Given this, how does any user know anything at all about another user of the system? For example, in Yenta, how does a user know that the person on the other end of some link is not his or her supervisor, romantic partner, or family member, trolling for interests that the user would rather not admit to? This is an example of the more general problem of spoofing—some user pretending to be someone else.

In general, this is a difficult problem. We shall sketch out our overall approach to it here, but many of the details must wait until Chapter 3 provides essential background and algorithms.

The architecture we present attempts to solve this problem by using reputations. Users may make any number of statements about themselves, called *attestations*, which are cryptographically signed by *other users* via their agents. These attestations are associated with the user's pseudonym—their Yenta-ID in Yenta, for example—and not their real identity, which may be unknown even to the user's own agent. It is beyond the scope of this architectural description to specify exactly how these other users acquire the trust to sign someone's attestation—in many cases, such as inside an organization, the users may be known to each other and therefore may sign each other's attestations on the basis of this shared knowledge. In other cases, such trust may come from long association and interactions through the application.

The web of trust

When two agents communicate, they may trade attestations. A user attempting to verify an attestation, whom we will call the *verifier*, must examine the signatures associated with the attestation, and must either convince himself that someone known to the user is one of the signatories, or that one of the signatories themselves has been

endorsed (via *their* signed attestation) by someone known to the verifier. The verifier is therefore attempting to construct a chain of signatures which terminates at one or more other users already known to the verifier. This tactic is exactly the same as is used to verify the identity corresponding to PGP public keys [187], and is called a *web of trust.* The details of how identities are handled, and the cryptographic algorithms used to sign attestations, are deferred to Chapter 3.

Verifying attestations is a fundamentally peer-to-peer operation. There is no *trusted certifying authority,* and no assumed hierarchy to the signatures being presented. How many signatures, from whom, and the exact structure required of the signature chain is completely up to the verifier's discretion. The verifier's policy may change depending on the use to which the information will be put—for example, in Yenta, a conversation to some unknown other user about a noncontroversial topic may not require any verification at all.

Word-of-mouth reputations

Like the referral algorithm described in Section 2.8, this is a word-of-mouth approach. It resembles the stereotype of small-town gossip and reputations, although this analogy is not exact—in small towns, the gossip is usually about third parties, whereas here the statements made are about the person who is making the statement.

There is nothing preventing a single *distinguished signer*—some signer that is well-known to a large fraction of users—from becoming established. This requires only that all users know about this signer, and that they trust it. Such a scenario is likely in an organization, which may have designated some individual to hold corporate cryptographic keys or the like, and which can disseminate to all users, through some mechanism not specified here, who the signer is and why the other users should trust it. However, such a distinguished signer is outside the scope of this architectural description; it is a local policy issue.

Any given user's attestations are stored (and offered) by his or her own agent. This must be so, because there is in general no distinguished location in the system to ask about any other user's reputation—the attestations come from the user himself. Because the user owns his own attestations, it is likely that only positive attestations, e.g., those that cast the user in a favorable light, will be offered. Verifiers thus walk a fine line in their judgments about attestations: while excessively positive attestations are unlikely to be signed by anyone trustworthy, negative attestations are unlikely to exist at all.

Additional details about the cryptographic operation of attestations is provided in Chapter 3. Yenta's use of attestations is described in Chapter 4.

2.12 Running multiple agents on one host

The architecture presented here has a rather unusual problem, namely, *how can multiple users run the application simultaneously on the same host?* At first glance, this appears completely straightforward—isn't it common that users on a timesharing host can both run telnet at the same time, for example?—but there are wrinkles in this architecture that make the straightforward solution inappropriate.

Typical client/server

Applications which use IP networks to communicate identify the connection via a 4-tuple of the local and remote host IP addresses and port numbers. In general, the host IP address determines *which computer* is involved, and the port number determines *which program* is involved, at each end of the link. Typical applications, such as telnet, depend on contacting a *known port* on the server end—for example, telnet uses port 23. A *daemon process* that listens to that port then creates an appropriate *server* which handles a client's inbound connection.

Privileged daemons

Unfortunately, this process requires that the daemon run as a *privileged user* under most operating systems, since it must be able to create the server process *as the*

appropriate user—otherwise, the server process could not access things that the user himself could access. If the server process was FTP, for example, the user would be unable to access his files unless everyone could.

Ephemerality of servers

Further, the server process that is created by this mechanism typically interacts only with the host operating system—its files and so forth—but does not then open additional network connections. Finally, server processes tend to be *ephemeral*—when the client network connection vanishes, so should the server.

We have different requirements

The architecture presented here is somewhat different. It is inconvenient to require that users running Yenta, say, also arrange to have their administrator install a privileged program in order to do so. Furthermore, such a privileged program would be tempting source for attack. For example, if all traffic passed through the daemon, it is potentially tappable at that point. And applications which use SSL to protect their communications—as Yenta does, for example (see Section 4.8.1)—cannot tunnel their encrypted data through the server, since the SSL architecture [63] does not permit this.

The portmapper

Instead, we run a *port mapper* service. The first copy of the application to be started on any given host starts listening on the *well-known-port—the WKP*—for the application. (In Yenta, for example, this is port 14990.) We shall call this copy of the application the *portmapper.* The portmapper's acquisition of the well-known-port prevents any other program on the system from listening on that same port. The application then *forks;* the other half of the fork then starts up as usual and runs the normal user application.

Acquiring the well-known-port; registering with the portmapper

Whenever any application starts up on the host, it attempts to acquire the WKP. If it succeeds, it forks as above, and one half becomes the portmapper. If it fails, then it knows that a portmapper is already running. In this case, the application scans the available range of ports until it finds one that is unused, and acquires it; let us call this port *P.* The application then *registers* with the portmapper—it gives the portmapper its *identity* (in Yenta, its Yenta-ID—see section 3.4) and the port it acquired. The portmapper stores this value in an internal table.

Inbound connections

Any inbound application attempts to connect on the well-known port. It specifies the *identity of the desired agent* that it wishes to communicate with—as above, in Yenta, this is the YID. The portmapper consults its internal table and tells the inquiring application to reconnect on port *P* instead.

Handling crashes

Applications try to reacquire the WKP at regular intervals. A success means that the existing portmapper must have died; the application that reacquired the port forks and becomes the new portmapper. Similarly, applications attempt to reregister with the portmapper at regular intervals; this enables a newly-started portmapper to rebuild its table.

Denial-of-service

A portmapper which acquires the port and then refuses to serve any requests—or which provides incorrect data for requests—is engaging in a denial-of-service attack; as we specified in Section 2.3, this is explicitly not a part of our threat model. (Presumably, on a real timesharing host, other users of the application will list the system's processes, discover the true identity of the user running the malicious portmapper, and will complain vigorously to the perpetrator.)

Security preserved

Note carefully how this approach fulfills the goals required of our architecture. The portmapper contains no personal data—agent ID's are public information. No personal data goes to any third-party process—the portmapper never sees the encrypted data stream between any two applications. No privileged process is required, and there is no single point at which security may be compromised.

2.13 Evaluation hooks

Our final topic of this chapter concerns *monitoring the operation of the system.* The sample application described in Chapter 4 is a research prototype, and consequently it is valuable to have the ability to collect information from it while it runs. Other applications might also benefit from the ability to observe their operation; such observation can be invaluable for locating architectural or implementation bugs, for example.

In arranging such a monitoring capability, however, we must be careful not to undo the privacy protections that the architecture tries so hard to put in place. The sketch that follows details some of the steps involved, so as to complete our architectural description. Details of how Yenta arranges to be monitored are presented in Chapter 4.

We assume that monitoring the running system can be accomplished by collecting statistics, from each agent, which detail what actions that agent has taken recently, whether or not it has detected any internal inconsistencies, and some information about its internal databases. Exactly what this information consists of is, of course, application-dependent.

A central receiver—a big problem?

In order to allow these statistics to be analyzed, they must be accumulated in a single place—a *central receiver* of statistical data. This is an alarming suggestions to anyone who has read Section 1.5: such a suggestion could potentially run afoul of all the problem of trust expressed in that section.

The key is to arrange for *anonymity of the collected data* and *confidentiality of its transmission.* We shall examine these in turn.

Anonymity

In order for the data to be *anonymous*, there must not be anything in it that can be related back to a particular user. We already assume that there is more than one user in the system, from Section 2.2, which makes the most obvious attack—knowing that *all* the data is from the system's only user—infeasible. The particular application being run must also take care to *sanitize* its data, by removing as many personally-identifiable details from the reported data as possible. For example, if the application handles messages between users, and it is important to see some of the contents of these messages, the identities of the correspondents should not be transmitted. Preferably, the messages themselves should not be reported—if what we care about is, say, the average message length, then only the length of the message should be reported in the first place. This is analogous to the caution expressed in Section 1.5 about not collecting anything which you are not willing to have be the subject of a subpoena.

The point of sanitizing the data is to eliminate the issue of having to trust the central server. This means that the central server can leave the accumulated data in the clear, on disks which might be the subject of an intrusion or subpoena, without compromising users' privacy.

Unlinkability must be what we are protecting

It is *very important* that the sanitization process takes into account that some data is dangerous *regardless* of whether it can be associated with a particular individual. For example, data on how to build a nuclear bomb in one's backyard, using components from the corner hardware store, should presumably not be allowed to reside on the central server even if it is not possible to connect it with any particular person—the mere disclosure of the data itself, due to compromise of the server, could have disastrous consequences. Care is required of the application designer if data like this could be present in the system.

Sanitizing the data is part of the solution. In many cases, however, one might wish to analyze the behavior of particular agents over time. It must be possible to determine unambiguously which agent is which, but it is presumably irrelevant exactly *whose* agent is the one reporting a particular item. In other words, we care about *distinguishing* agents from each other, but not in *mapping* them back to user identities.

Random unique-ID's

The solution to this problem is straightforward—have each agent assign itself a unique identifier, not related in any way to anything else about the user (neither the user's identity, nor his characteristics), and report that unique identifier when sending data to the central receiver. This identifier should *not* be the same as the identifier which is a pseudonym for the user—or any other identifier at all—since the whole point is to make statistical data collection unlinkable to actual users or their online identities. For example, in Yenta, the ID we are discussing here is *not* the Yenta-ID. This unique identifier can be simply any sufficiently-random collection of bits which is long enough that accidental collisions (birthday paradoxes) are unlikely. For example, in any reasonable application, 128 bits is perfectly sufficient.

If the data is sufficiently sanitized before transmission, and any identification information is restricted to disambiguating multiple agents from each other, then the data as collected at the central server is relatively safe. None of the threats mentioned in Section 1.5 present an insurmountable problem, because the data cannot be related back to anyone who could be harmed by its disclosure, and we are assuming that the data collected is inherently safe if its source is unknown.

Confidentiality

The remaining issue is *confidentiality.* It is insufficient to protect the data only once it arrives at the server, since an eavesdropper may be present between any given agent and the server. (Indeed, one of the best places such an eavesdropper could possibly be is right at the server, since *all* application traffic destined for the server will pass that point.) Such an eavesdropper could identify both the contents of the traffic and, for instance, the IP address of its origin; this could lead to disclosure of the mapping between any particular piece of data and the user who originated it.

To protect users against this threat, the data in transit to the central server must be encrypted.

Unless the application logs at different intervals or at different lengths depending on some confidential data, or unless the mere fact that a given user is running the application at all is considered confidential, this is sufficient to defeat eavesdropping of the contents of the transmission, and traffic analysis of the communication.

Note that if merely whether or not someone is running the application is considered confidential, we may use a modification of the broadcasting solution of Section 2.10 to help. Rather than having every agent log directly to the central server, it could ask that its logging information be routed through n random other members of some cluster(s) before final transmission. The intermediate hops need not (indeed, cannot) decrypt the communication, and the central server (and any eavesdropper positioned there) has no idea where the logging information truly originated. If we are using this tactic, then the actual encrypted data should be encrypted with a public key whose corresponding private key is known only to the central server, and not to any agent in the system. Intermediate agents cannot then decrypt the data, and even an eavesdropper at the server who possesses the server's private key cannot, by the time the data is received, know where it came from.

Central server is not a fundamental part of the architecture

It should again be emphasized that the rest of the architecture presented in this chapter does *not* depend in any way on the existence of a central collector of statistical data. Such a capability, while valuable for debugging or research, need not necessarily be in any deployed application. Indeed, one can make arguments that a system which is *not* the subject of research or debugging should not run such a server. It represents a potential source of privacy violations for its users, and also represents a potentially large source of inbound traffic for whatever network site hosts it.

Robustness vs logging

Also, it should be pointed out that *robustness* issues imply that agents which wish to log information to the central server should *fail gracefully* if the server is unavailable.

They should potentially queue data for later delivery, but should not hang if the central server is not always available, and should not maintain this queued data indefinitely in any case, or their storage may grow without bound. This keeps the system as a whole from freezing if the central server is temporarily or permanently taken offline, and keeps storage on local agents from growing monotonically as well.

2.14 Summary

In this chapter, we have examined the basic elements of the architecture. We have discussed what traits are shared by applications for which the architecture was designed, and which problems we do not address. We have briefly described the sample application which has been implemented to test the architecture, and extensively described those elements of the architecture which support it, including how agents may cluster, how the resulting groups may be used, the reputation system, and how evaluation data may be safely collected.

CHAPTER 3 *Privacy and Security*

3.1 Introduction

This chapter addresses privacy and security concerns in the architecture we described in Chapter 2. It assumes knowledge of the contents of that chapter, but not necessarily in-depth knowledge of modern cryptography or computer security.

We shall describe:

- The nature of the problem, including: Section 3.2
  - The threat model, such as what attacks we expect, and the difference between passive and active attackers Section 3.2.1
  - A discussion of how private we are trying to be Section 3.2.2
  - Some desiderata for security design in general, and how our architecture makes use of them Section 3.2.3
  - The problems we are *not* attempting to solve Section 3.2.4
- Some useful cryptographic techniques, including: Section 3.3
  - Symmetric encryption Section 3.3.1
  - Public-key encryption Section 3.3.2
  - Cryptographic hashes Section 3.3.3
  - Some issues in key distribution Section 3.3.4
- Some of the solutions we employ, including: Section 3.4
  - How anonymity and pseudonymity help Section 3.4.1
  - Various techniques against passive attackers Section 3.4.2
  - Various techniques against active attackers Section 3.4.3
  - Issues involved in protecting the distribution Section 3.4.4
- Selected additional topics which tie up some loose ends Section 3.5

3.2 The problem

This section discusses the types of attacks the architecture is likely to see, as well as the problems we are *not* trying to solve.

3.2.1 The threat model: what attacks may we expect?

Given the architecture described in the previous chapter, there are a wide variety of potential attacks which may be mounted by malicious or curious third parties. They generally break down into *passive* attacks, in which communications are merely mon-

itored, and *active* attacks, in which communications or the underlying agents themselves are subverted, via deletion, modification, or addition of data to the network.

Packet sniffing

Passive attacks. The most obvious attack is simple monitoring of packet data; such an attack is often accomplished with a *packet sniffer*, which simply records all packets transmitted between any number of sources. If such data includes users' mail messages or files, then two agents which are trading this information back and forth will leak information to an *eavesdropper.*

Traffic analysis

Even if the actual communications between agents are perfectly encrypted, however, passive attacks can still be quite powerful. The easiest such attack, in the face of encrypted communications, is *traffic analysis*, in which the eavesdropper monitors the *pattern* of packet exchange between agents, even if the actual *contents* of the packets are a mystery. This can be surprisingly effective: It was traffic analysis that alerted a pizza delivery service local to the Pentagon—and thus the media—when the United States was preparing a military action at the beginning of the Gulf War; when late-night deliveries of pizza suddenly jumped, it became obvious that something was up [181]. (Even though [179] points out that press coverage of the pizza effect tends to quote unnamed sources, a very small number of individuals with personal stakes—such as a Domino's manager in the area—and other press reports, the continuing press coverage [151] and even military recommendations [174] surrounding such effects make it clear that this threat is taken seriously.)

Spoofing and replays

Active attacks. Active attacks involve disrupting the communications paths between agents, or attacking the underlying infrastructure. The most common such attack is a *spoofing* attack, in which one agent impersonates another, or some outside attacker injects packets into the communication system to simulate such an outcome. Often, spoofing is accomplished via a *replay* attack, in which prior communications between two agents are simply repeated by the outsider. Even if the plaintext of the encrypted contents of the communication are not known, such attacks can succeed so long as duplicate communications are allowed and the attacker can deduce the effect of such a repeat. For instance, if it is noticed that a cash-dispensing machine will always dispense money if a particular (encrypted) packet goes by, a simple replay can spoof the machine into disgorging additional cash.

Subverted agents

More sophisticated attacks are certainly possible. Individual running agents might be subverted by a third party, such that they are no longer trustworthy. Such a subverted agent might use encryption keys which are known to the interloper, for example. Alternately, the attacker might create his or own own agent, which looks like a genuine agent to the rest of the network, but pretends to have characteristics which match *everything*—in Yenta, for example, such an agent might then be used to troll for people interested in particular topics, and presumably also would be modified to disgorge anything interesting to its creator.

Subverted distribution

Finally, the actual distributed agent might be modified by a determined attacker at the source itself—say, by subtly introducing a trojan horse into the application at its distribution point(s), either by modifying its source code, or by modifying any precompiled binaries which are being distributed. This is essentially a more-distributed and more-damaging version of the subverted-agent attack above. As an example, consider all the Web pages currently extant which proclaim, "These pages are best viewed with Netscape *x.y.* Download a copy!" Now imagine what would happen if the link pointed to a carefully-modified version of Netscape that always supplied the *same* session key, known to the interloper: the result would be that anyone who took the bait would be running a version of Netscape with no security whatsoever, hence leaving themselves vulnerable to, e.g., a sniffing attack on their credit card number.

3.2.2 How private is private?

Consider the degrees of anonymity offered by the chart below:

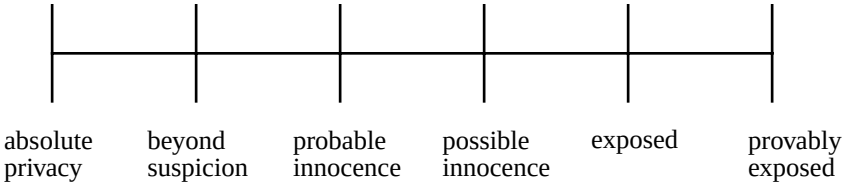

Figure 3: Degrees of anonymity

As defined in [141], the extremes of this chart range from *absolute privacy*, where the attacker cannot perceive the presence of communication, to *provably exposed*, where the attacker can prove the sender, receiver, or their relationship to others. The discussion advanced in [141] is oriented more towards the perception of communication *at all*, whereas we are concerned with the *contents* of that communication as well, but the spectrum of possibilities is nonetheless useful. They define the rest of the chart as follows:

- A sender is *beyond suspicion* if, though the attacker can see evidence of a sent message, the sender appears no more likely to be the originator of that message than any other potential sender in the system.
- A sender is *probably innocent* if, from the attacker's point of view, the sender appears no more likely to be the originator than to not be the originator. This is weaker than beyond suspicion in that the attacker may have reason to expect that the sender is more likely to be responsible than any other potential sender, but it still appears at least as likely that the sender is not responsible.
- A sender is *possibly innocent* if, from the attacker's point of view, there is a non-trivial probability that the real sender is someone else. While weaker than the above, it may prevent attackers from *acting* on their suspicions.
- Finally, a sender is *exposed* if an attacker can see information which unambiguously identifies the sender. *This is the default for almost all communications protocols on the Internet*—most such protocols are cleartext, and make no attempt to hide the addresses of senders or receivers. This is weaker than being *provably exposed*, however, since it is generally the identity of the *computer* that is revealed, rather than some nonrepudiable *user* identity.

The architecture discussed here, for the most part, attempts to ensure either *possible innocence* or *probable innocence*; we shall differentiate where useful. In addition, certain parts of the architecture may make it possible for the user to be *beyond suspicion* to a *local* eavesdropper—someone who can monitor some, but not all, communication links in the system.

3.2.3 Security design desiderata

The security architecture presented here is cognizant of several principles which are well-known in the security and cryptographic communities. This section discusses several of them, and demonstrates how they have motivated various decisions taken in the design.

Design must be open—the importance of open source

Security through obscurity of design does not work. This means that any design which depends upon secrecy of the design is guaranteed to fail, since secrets have a way of getting out. Since this architecture is designed to be run by a large number of individuals all across the Internet, its binaries must be public, hence security through obscurity would be untenable anyway in the face of disassemblers and reverse-engineering. (In fact, the source code of Yenta, the sample application, is also public, which should increase confidence in the resulting system; see the discussion of Yvette in Section 3.4.4.)

Protect the keys

Keys are the important entity to protect. In good cryptographic algorithms, it is the keys that are the important data. Since keys are usually a small number of bits—hundreds or perhaps thousands at most. Because new keys are often trivial to generate, protecting keys is much easier than protecting algorithms. Unfortunately, however, *key management*—keeping track of keys and keeping them from being accidentally disclosed—is often the hardest and weakest point of a cryptosystem [6][15][24][34][69]. Our architecture has a variety of keys and manages them carefully.

Use existing crypto

Good cryptography is hard to design and hard to verify. Most brand-new cryptographic systems turn out to have serious flaws. Only when a system has been carefully inspected by a number of people is it reasonable to trust it. This is another reason why security through obscurity is a bad idea. We depend on well-established algorithms and protocols for our fundamental security, since they have been carefully scrutinized.

Whole-system design

Security is a function of the entire system, not individual pieces. This means that even good cryptography and system design is worthless if it can be compromised by bribing or threatening someone. Part of the reason for the decentralized nature of this architecture is to avoid having a single point of compromise, as detailed in Chapter 1.

Poor design is dangerous too

Malevolence and poor design are sometimes indistinguishable. Many system failures that look like the result of malevolence are instead the result of the interaction of an accident and some unfortunate element of the design. For example, the entire ARPAnet failed one Sunday morning in 1973 due to a double-bit error in a single IMP [121]. A similarly disastrous outcome from a simple, single error is aptly described by this quote: "The whole thing was an accident. No saboteur could have been so wildly optimistic as to think he could destroy an airplane this way," which described how an aircraft was demolished on a friendly airfield during World War II when someone ingeniously circumvented safety measures and inadvertently connected a mislabelled hydrogen cylinder to the plane's oxygen system [137].

Minimize collected information

If you don't want to be subpoenaed for it, don't collect it. As we mentioned in Chapter 1, Federal Express, a delivery service in the United States, receives (and hence is compelled to respond to) several hundred subpoenas a day for its shipping records [178]. The safest way to protect private data collected from others from such disclosure—not to mention the hassle of responding to a stream of subpoenas—is never to collect it in the first place. Both the lending records of most libraries, and the logfiles of MIT's primary mailers—which are guaranteed to be thrown away irretrievably when three days old [153]—adhere to this rule. This also motivates our decentralized design: any central point is a subpoena target.

We can't be perfect

Security is a spectrum, not an absolute. A computer can often be made perfectly secure by unplugging it—not to mention vaporizing its disks—and their backups. However, this is a high price to pay. Tradeoffs between security and functionality or performance are often necessary. It is also true that new attacks are constantly being invented; hence, while this research aims at a more-secure implementation than that which is possible without attending to these issues at all, we can never claim to be completely secure. We therefore aim for security that is good enough, and to *do not harm*—such that user privacy is protected as well or nearly as well as it would be if the application was not running. We cannot hope for better—doing better would imply that our application somehow magically improves the security of other, unrelated applications—and may have to make *some* tradeoffs that nonetheless lead to a little bit of insecurity for a large benefit.

3.2.4 Problems not addressed

There are a number of problems which are *not* addressed in the security architecture presented here. The problems we are not addressing influence where we will and will not accept design compromises.

No mobile code

For instance, since each agent runs on a user's individual workstation, and each agent is not itself a mobile agent per se [29][35][70][170][183], we do not have the problem of executing arbitrary chunks of possibly-untrusted code on the user's local workstation.

No Byzantine failures

Further, it is assumed that, while *some* agents may have been deliberately compromised, the vast majority of them have not. This mostly frees us from having to worry about the problems of *Byzantine failure* [53][131] in the system design, wherein a *large* portion of the participants are either malfunctioning or actively malicious.

We also assume, as in the Byzantine case, that not *every* other agent any *particular* agent communicates with is compromised. If this were not true, certain parts of the algorithm would be vulnerable to a *ubiquitous* form of the *man-in-the-middle* attack, wherein an interloper pretends to be A while talking to B, and B while talking to A, with neither of them the wiser. (Weaker forms of this, wherein there are only a *few* agents doing this, have reasonable solutions. In general, when dealing with Byzantine failures, the amount of work to cope with increasing numbers of hostile peers goes up quite rapidly—exponentially in many cases. This means that dealing with a small number of miscreants is feasible, whereas the situation where most peers are untrustworthy becomes very difficult.)

Trusted path to binaries

The architecture provides no protection for the user if his or her copy of the application has been compromised. It is generally trivial for a sophisticated attacker to compromise a binary—for example, by substituting NFS packets on the wire as the application is loaded from the fileserver. We cannot be of any help in this case; a user without a trusted path to his or her binaries is already at the mercy of any good attacker, regardless of the application being run.

Cracked root

Along the same lines, a user who runs the application on untrusted hardware cannot expect that it can never be compromised—this is analogous to not having a trusted path to one's binaries, since an attacker who has compromised the computer on which the application is being run can by definition either read or alter data in the running binary. Consider the example of Yenta, which runs as a *daemon* and remains resident in memory indefinitely. The user's secret key, which is the basis of his or her identity, must similarly remain in memory for long periods of time. If this were not the case, then the user would have to constantly type his or her passphrase for *every* operation which required an identity check, of which there are many. But this also means that any attacker who has root access to the user's workstation, for example, can read this key out of the process address space. Hence, if the user's workstation has poor security in general, then Yenta's ability to keep the user's secrets from the attacker will be no better.

Poor passphrases

While this architecture tries to minimize the number of places where users can inadvertently compromise their own security, some user responsibility is nonetheless expected. For example, the agent must store its permanent state somewhere. If this data is to be private, it must be protected. Absent hardware solutions, the most reasonable solution to this protection is to encrypt it with a passphrase—but nothing can help us if the user chooses a poor passphrase, such as one that is too short or is easily guessed.

Government agencies or rubber-hose cryptanalysis

Similarly, this architecture is no protection against the resources of a government agency, or some similarly-equipped adversary. Such an adversary has no reason to attempt a subtle compromise of the distribution, the protocols, or the cryptography. It

may instead physically bug the user's premises, compromise his hardware, or use *rubber-hose cryptography*—coercing the user's key(s) via implied or explicit threat of physical force. A possible solution to coercion is the use of *deniable filesystems* [22], but this is beyond the scope of the research presented here.

No denial-of-service

In addition, we do not explicitly deal with *denial-of-service* attacks, which are extremely difficult for any distributed system to address. Such attacks amount to, for example, dropping every packet between two agents which are trying to communicate—this attack looks like the network has been partitioned to the agents involved, and there is little defense.

No international export

Finally, we have the problem of our use of strong cryptography to protect users' privacy. The United States government currently regulates such cryptographic software as a munition, under *EAR*, the Export Administration Regulations [50]—formerly *ITAR*, the International Treaty On Arms Regulations [87]. This means, for example, that the cryptographic portions of Yenta's software are currently unavailable outside the US unless added back in elsewhere. Solving the limitations of EAR/ITAR is not explicitly addressed here—except to demonstrate how such governmental policies work against the sovereign rights of its citizens, as we detail in Chapter 1.

3.3 Cryptographic techniques

This section introduces some useful cryptographic techniques that will be used later. The techniques we discuss are used as *black boxes*, without proof that they properly implement the functionality described for the box and without the mathematical background which underlies them; those who wish to check these assertions may examine the citations where appropriate. In particular, for a much more complete introduction that includes an excellent survey of the field, see [155].

3.3.1 Symmetric encryption

One of the most straightforward cryptographic techniques uses *symmetric keys*. Algorithms such as IDEA ([155] pp. 319-324) work this way. Given a 128-bit key, the algorithm takes *plaintext* and converts it to *ciphertext*. Given the same key, it also converts ciphertext back into plaintext. Expressed mathematically, we can say that C=K(P) [the ciphertext C is computed from the plaintext P via a function of the key K], and similarly P=K(C) [the reverse also works].

IDEA is probably very secure. The problem comes in *distributing the keys*: we cannot just transmit the keys before the encrypted message—after all, the channel is deemed insecure or we wouldn't need encryption in the first place—hence users must first meet *out-of-band*, e.g., not using the insecure channel, to exchange keys. This is infeasible for a large variety of applications.

3.3.2 Public-key encryption

A better approach uses a *public-key cryptosystem* [PKC], such as RSA ([155] pp. 466-473) or the many other variants of this technology. In a public key system, each user has *two* keys: a *public* key and a *private* key, which must be generated together—neither is useful without the other. As its name implies, each user's public key really is public—it can be published in the newspaper. The private key, on the other hand, is *never* shared, not even with someone the user wishes to communicate with.

Confidentiality

User A encrypts a message to B by computing $C=K_{PB}(P)$, e.g., a function involving B's *public* key. To decrypt, B computes $P=K_{SB}(C)$, e.g., B's *private* key. Note that, once encrypted, A *cannot* decrypt the resulting message, using any key A has access to—the encryption acts *one-way* if A does not have B's private key—and she shouldn't! [One important detail: since PKC's are usually slow, one usually creates a brand-new *session key*, transmits *that* using PKC, then uses the session key with a symmetric cipher such as IDEA or triple-DES to transmit the actual message. In addition, PKC's may sometimes leak bits if used to encrypt large amounts of data; encrypting only keys can avoid this problem.]

Authenticity

This scheme provides not only *confidentiality*—third parties cannot read the messages—but also *authenticity*—B can prove that A sent the message. How does this work? Before A sends a message, she first *signs* the message by encrypting it (really a *cryptographic hash* of the message—see below) with *her own private key.* In other words, A computes $P_{signed}= K_{SA}(P)$. Then, A *encrypts* the message to B, computing $C=K_{PB}(P_{signed})$. B, upon receiving the message, computes $P_{signed}=K_{SB}(C)$, which recovers the plaintext, and can then *verify* A's signature by computing $P=K_{PA}(P_{signed})$. B can do this, because he is using A's *public* key to make the computation; on the other hand, for this to have worked at all, A must have sent it, because only her *private* key could have signed the message such that her public key worked to check it. Only if someone had cracked or stolen A's private key could the signature have been fraudulently created.

3.3.3 Cryptographic hashes

It is often the case that one merely wishes to know whether some message has been tampered with. One obvious solution is to transmit the message *out of band*—via some channel which is not the same as the channel originally used to transmit the message. But this begs the question of how *that* channel is secured, and can be very inconvenient to implement in any case.

An easy way to avoid out-of-band transmission is via a cryptographic hash, such as MD5 ([155], pp. 436-441) or the Secure Hash Algorithm (SHA, [155], pp. 442-445). These hash functions compute a short (128-bit or 160-bit, respectively) message digest of an unlimited-length original message. These functions have the unusual property that changing any single bit of the original message changes, on average, half of the bits of the digest. Further, they function in a one-way fashion—it is infeasible, given a digest, to compute a message which, when hashed, would yield the given digest.

On the other hand, anyone can compute the hash of a message, since the algorithm is public and uses no keys. This means that it is computationally easy to verify that a particular message does, in fact, hash to a particular value, even though it is infeasible to *find* a message which produces some particular hash.

Digital signatures

Since such hashes are compact yet give an unambiguous indication of whether the original message has been altered, they are often used to implement *digital signatures* such as in the RSA scheme above—what is signed is not the actual cleartext message, but a hash of it. This also improves the speed of signing (since signing a 128- or 160-bit hash is much faster than signing a long message), and the actual security of the cipher as well (because RSA is vulnerable to a *chosen-plaintext* attack; see [155], p. 471).

3.3.4 Key distribution

One of the hardest problems of most cryptosystems, even public-key systems, is correctly *distributing* and *managing* keys. In a public-key system, the obvious attacks—compromise of the actual private key—are often relatively easy to guard against: keep the private key in memory as little as possible, encrypt it on disk using DES with a passphrase typed in by the user to unlock it [187], and keep it offline on a floppy if possible.

But consider this: Alice wishes to send a message to Bob. She looks up Bob's public key, but interloper Mallot intercedes and supplies *his own* public key. Alice has no way of knowing that Mallot has done so, but the result of her encryption is a message that *only* Mallot, and not Bob, can read! Even if one demands that Alice and Bob have a round-trip conversation to prove that they can communicate, Mallot could be playing man-in-the-middle, simultaneously decrypting and re-encrypting in both directions as appropriate.

Webs of trust

To solve this problem, systems such as Privacy Enhanced Mail [92] use a centralized, tree-structured key registry, which is inconsistent with our decentralized, no-hierarchy architecture. On the other hand, PGP [187] functions with completely decentralized keys, by having users *sign each other's keys*—this is the same mechanism used in the attestation system described in Section 2.11. When Alice gets "Bob's" public key, she checks its signatures to see if *someone she trusts* has signed that key, or some short chain of trustable people, etc. If so, then this key must be genuine (or there is a conspiracy afoot amongst their mutual friends); if not, then the key may be a forgery. This practice of signing the keys of those you vouch for is called the *PGP web of trust* and is the primary safeguard against forged keys. Yenta, for example, uses this technique in *signing attestations* as part of its reputation system.

3.4 Structure of the solutions

This section presents solutions to some likely security problems in our architecture, using some of the technology mentioned previously. It presents a *range* of solutions; not every user in every application might want the overhead of the most complete protection, and the elements, while often solving separate problems, sometimes also act synergistically to improve the situation. Finally, for brevity, it omits some details present in the complete design.

3.4.1 The nature of identity

Uniqueness and confidentiality. It should not be possible to easily spoof the identity of an agent. For this reason, every agent sports a unique *cryptographic identity*—a *digital pseudonym*. This identity corresponds, essentially, to the *key fingerprint* [187] of the individual agent's public key—a short (128 bits) cryptographic hash of the entire key. In Yenta, this identity is referred to as the user's *Yenta-ID* or *YID*, and is effectively a random number—knowing it does not tell anyone anything about whose *real-life* identity it is. In order to keep some interloper from stealing, say, agent A's pseudonym, any agent communicating with A encrypts messages using A's public key. A can prove that its pseudonym is genuine by being able to decrypt; further, such communications are *interlocked* [155] and have an internal sequence number—itself encrypted—both of which help prevent replay attacks by a man in the middle. Further, of course, such encryption prevents an eavesdropper from intercepting the actual conversation. Thus, even though the actual identity of the user is not known, the user's pseudonym cannot be appropriated.

Pseudonymity and anonymity are cornerstones of the design

The fact that users are by default pseudonymous, and often completely anonymous, is a critical aspect of the security of the architecture. Consider, for example, what would happen if characteristics that were offered during clustering (Section 2.8.2) automatically identified the user identity that went along with them—the user would be *exposed*, in the terminology of Section 3.2.2. Instead, given the *plausible deniability* feature described in Section 2.8.2, it is at least possible that any given characteristic does not correspond to the agent offering it, meaning that the user is *probably*, and at least *possibly, innocent.* In addition, if third-party subcomparisons and random-reforwarding via cluster broadcasts, also as described in Section 2.8.2, are in use, the user may well be *beyond suspicion.*

The completely decentralized nature of our architecture complicates key distribution. The model adopted is the decentralized model used by PGP [187]. By not relying on a central registry, we eliminate that particular class of failures. And interestingly, the architecture partially eliminates the disadvantage of PGP's decentralized key distribution—that of guaranteeing that any particular public key really does correspond to the individual for which it is claimed. In PGP, we care strongly about actual individuals, but in our architecture, and in the sample application, only the cryptographic ID's are important—for example, Yenta tries to *hide* the true identity of its users unless they arrange to be known to each other.

Spamming and spoofing. Unfortunately, this pseudonymity comes at a price: For example, if we are about to be introduced to some user, how can we have any idea who we might be about to be introduced to? Can we know that the last 10 agents we've seen do not all surreptitiously belong the same individual? Can a given user of Yenta, for instance, know that this person won't spam us with junk mail once he discovers our interest in a particular topic? And so forth.

We solve this problem with the *attestation system*, described in Section 2.11. This system provides a set of reputations, which are useful in verifying, if not the identity of a given user, at least whether he or she can make statements about himself or herself that other users will vouch for.

3.4.2 Eavesdropping

The generally-encrypted nature of inter-agent communication makes most eavesdropping, including some but not all man-in-the-middle attacks, quite difficult. However, traffic analysis is still a possibility—for example, if an interloper knows what one Yenta is interested in, watching who it clusters with could be useful.

Fortunately, we have a solution to this, in the broadcasting paradigm mentioned in Section 2.10. In addition, we can use the techniques in Section 3.4.3 to provide additional security.

3.4.3 Malicious agents

If some malicious person was running a subverted version of an agent, what could he discover? The most important information consists of the identities of other agents in the cluster cache—especially if those identities can be those of real users, e.g., their *real names*, and not digital pseudonyms—and the contents in the rumor cache—especially if, again, such text can be correlated to real people. There are therefore two general strategies to combat this: hiding real identifying information as well as possible, and minimizing the amount of text stored in the rumor cache. We shall mention two of the simplest approaches below; other approaches to both problems, involving Mixmaster-style random-reforwarding, secret-sharing protocols, or diffusing pieces of characteristics out to large numbers of third parties for comparison, are possible but are more complicated than necessary for this discussion.

Hiding identities

Since users are pseudonymous by default, hiding their identities in large part centers around avoiding traffic analysis. Using the *broadcasting* strategies presented above suffices. For a more complete description, please see Section 2.10.

Mixing in other agents' data

A simple technique for protecting users' characteristics against possibly malicious agents is to *mix in other agents' data* when engaging in the comparison and referral process. For a more complete description of this process, please see Section 2.8.3.

A range of privacy is available

Depending on which of the strategies above are chosen, and the nature of the characteristics handled by the application, it may be possible to arrange several degrees of user privacy. Using the terminology of Section 3.2.2, these could plausibly range from *possible innocence* to *beyond suspicion*.

3.4.4 Protecting the distribution

There is a final piece of the puzzle—how do users of an agent know that their copy is trustworthy? The easiest approach, of course, is to cryptographically sign the binaries, such that any given binary may be checked for tampering with the authoritative distribution point. But what if the program itself, at the distribution point, had a trojan horse inserted into its *source*, either by the implementors themselves, or by a malicious third party who penetrates the development machine? Even though the source is freely distributed, and may be recompiled by end-users and checked against the binary, what individual user would want to read the *entire* source to check for malicious inclusions? This is, of course, a problem for *any* software, and not just agents in the architecture we present here—but applications such as Yenta are particularly diffi-

cult for a user to verify solely from their behavior. After all, they read sensitive files *and* engage in a lot of network traffic—and even worse, the traffic is encrypted, so one cannot even check up on it with a packet sniffer.

In general, those who distribute software have chosen one of three models:

- *Trust us.* Often used by those who do not provide source code at all.
- *Go ahead—read every line of the source code yourself.* This is an infeasible task for almost any reasonable application, and a huge burden.
- *Hope you hear something one way or the other on the net or in the press.* This, too, is both infeasible, error-prone, and subject to a variety of false positives and false negatives.

The Yenta code vetter

There is another way. To demonstrate this, we have developed *Yvette*, a Web-based tool which allows multiple people to *collaboratively* evaluate an agent's source code—in this case, Yenta's. A summary of Yvette's capabilities is presented below, and examples of its use are presented in Figure 4 and Figure 5.

Evaluators store cryptographically-signed—hence traceable and non-spoofable—comments on particular pieces of the source where others can view them. The signature covers both the comment *and* the exact text of the code being commented upon. Each individual need only check a small piece of the whole, yet anyone can examine the collected comments and decide whether their contents and coverage add up to an evaluation one can trust.

Yvette presents an interface, via the Web, which allows anyone to ask questions such as:

- Who has commented on this particular piece of code? Are the comments mostly favorable, or not? What is the exact text of the comment(s)?
- What regions have the most or least number of comments associated with them?

Yvette users may also take actions such as:

- Download, for inspection and comment, a piece of the source, which can be a region of lines in a file, a subroutine, a set of subroutines, a set of files, or an entire directory tree.
- Upload cryptographically-signed comments about some piece of downloaded source code.

Note that, since it distributes code that may include cryptographic routines whose export from the US and Canada is illegal [50][87], Yvette must also be aware of which sections of code are sensitive and must use address-based heuristics and questions of the user—only for those parts of Yenta which are cryptographic—to ensure that EAR/ITAR's export restrictions [50][87] are not violated. The heuristics used are the same as those used to control the export of PGP [187], which, while easy to circumvent, are informally viewed as sufficient by at least some of the relevant players in the US government [104].

Using Yvette, therefore, users who wish to help verify a distribution can bite off a small piece of the problem, asking the Yvette server for which pieces of source code have not yet been extensively vetted, perusing other people's comments, and so forth. Users with no programming experience, but who nonetheless wish to check the distribution, may look at everyone else's comments to assure themselves of the integrity of the product.

Yvette thus attempts to encourage a *whole-system* approach to security, in which not only are the agents themselves secure, but their users—who are also part of the system—may easily *trust* the agents' security and integrity. It is hoped that mechanisms such as Yvette will become more popular in software distribution in general, and that it encourages thinking about more than just protocols and cryptography—if we expect widespread adoption of sophisticated agents, the sociology of how users can use and trust them matters, too.

3.5 Selected additional topics

There are a few loose ends in the architecture we present here that have not been adequately addressed by the discussion so far. This section attempts to tie them up.

Central servers

Let us consider first the central servers that exist in the design, namely the *bootserver* (described in Section 2.7) and the *statserver* (described in Section 2.13). Both of these servers are safe, in the sense of the unlinkability of users' personal data and their actual identities, but in slightly different ways.

The bootserver knows IP addresses. Because of this, it could potentially lead an attacker directly back to an individual. However, the bootserver knows nothing else—in particular, it knows nothing about any user's characteristics, save that the given user runs the application at all.

The statserver, on the other hand, potentially knows quite a bit about all users—in Yenta, for example, it knows information such as how many clusters the user is in, how they tend to use the user interface, what machine architecture Yenta is being run on, and so forth. (Note that it still does not know the detailed contents of individual characteristics, because such information could compromise a user's privacy if revealed, *and* it is unlikely to be so useful for analysis of Yenta's behavior that the risk is worthwhile.) However, the statserver does *not* know user identities or IP addresses at all. Once the data has been stored on disk, the agent's identity and IP address are gone. The only data that the statserver has preserved is a unique *random number* which can be used to differentiate one agent from another, but nothing else.

Encrypted connections

As for the safety of the data getting to the statserver in the first place, or between any given pair of agents, note that we have specified that *all* communications are routinely encrypted. The only exception is in data which contains no personal user data, namely bootstrap requests and replies, either via broadcast or to and from the bootserver. For details of how Yenta performs such encryption, see Section 4.8.1.

Persistent state

Getting the data between agents is only part of the story, however; we must also consider the storage of the agent's persistent state across shutdowns. In most applications, this is likely to stored be in a filesystem on a disk. If the agent handles personal information, this storage point is a tempting target for an attacker. Furthermore, it is likely that the application may store users' private keys—perhaps the basis of their identity—in this file as well, meaning that an attacker who can read the file can not only violate the user's privacy, but impersonate him or her to other users as well, with potentially serious implications.

It is clear, therefore, that such data should be protected. Exactly how this is to be accomplished is application- and implementation-specific; how Yenta does so is described in Section 4.8.2. Note in particular that this is a rich source of possible security problems, for several reasons:

- A network connection is necessarily a moving target—if an eavesdropper fails to intercept the relevant packets, the attack fails. On the other hand, data stored on disk is vulnerable to compromise from the moment it is created until long afterwards—perhaps even after it is thought deleted, and, to a sophisticated adversary, even after the disk has been formatted [73]. Keeping backups around forever, and

failing to adequately encrypt their contents or control physical access to them, only makes this worse.

- If the data is stored encrypted, we have the often-difficult problem of how to securely ask the user for the decryption key. Many environments provide no known-secure method of eliciting such data; in particular, UNIX users who use the X Window System [152] or Telnet [136] are particularly vulnerable to simple packet sniffing, and this is an extremely popular attack. While it is possible for knowledgeable users to use SSH [158] or Kerberos' ktelnet [127], there is often no way for the application to ensure this—and the consequences of a single instance of carelessness could lead to the user's privacy being unknowingly compromised forever afterwards.
- Encrypting the data with the same encryption key every time it is written to disk exposes it to a number of attacks if the data varies [155].

Random numbers

Finally, note that we have at many points mentioned the term *random number*—whether explicitly, in Section 2.13's discussion of the ID's generated for statserver, or implicitly, whenever we talk about generating any sort of key—session keys, public/private key pairs, and so forth. We assume here that such random numbers are really *pseudorandom*, e.g., derived from deterministic software.

Where are these random numbers coming from? Certainly not from typical application libraries; most random number sources provided with most operating systems are extremely poor when employed for cryptographic applications. Further, several high-profile examples of poor decisions in sources of random numbers have come to light, such as an early Netscape attempt at SSL [63] which derived its "random" numbers from easily-predictable values provided by the host operating system—this meant that a browser's supposedly-secure, 128-bit-key connection to a server could be broken in around 25 seconds [67].

Thus, the implementor must take great care in selection and use of random numbers in the application. This is common sense in cryptographic circles, but it bears repeating here. Exactly where to find a good source of randomness is always implementation-specific; some operating systems make available random numbers which are derived from turbulent processes (such as disk head performance), but many do not.

For an illustrative example of how Yenta acquires, manages, and uses random numbers, see Section 4.8.3.

3.6 Summary

In this chapter, we have described the threat model—what sorts of attacks we consider within the scope of this research. We then presented some background on modern cryptography and how it can help address many of the threats presented; we also discussed how decentralization of the architecture contributes greatly to the protection we afford. Finally, we presented a new method which makes collaboratively evaluating the source code of a critical application easier, and tied up some loose ends.

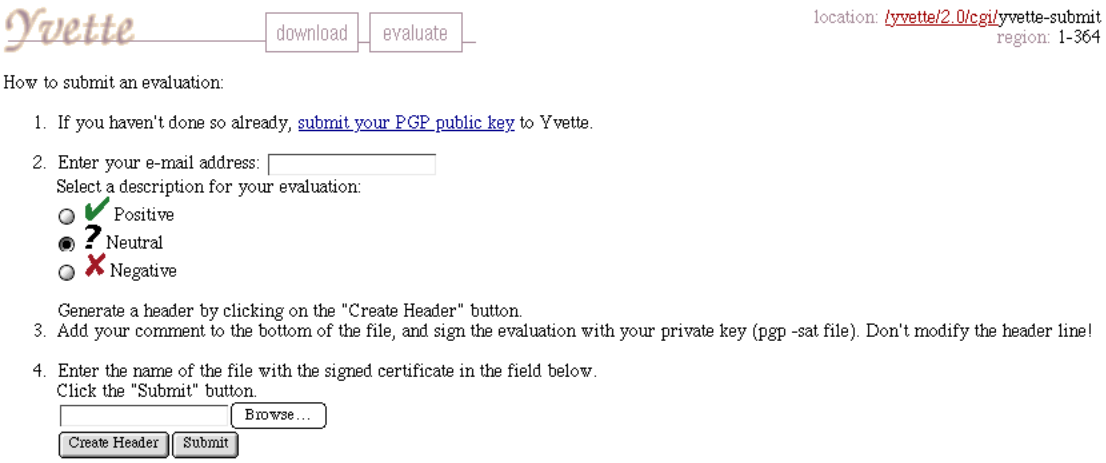

Figure 4: Showing the user how to submit an evaluation.

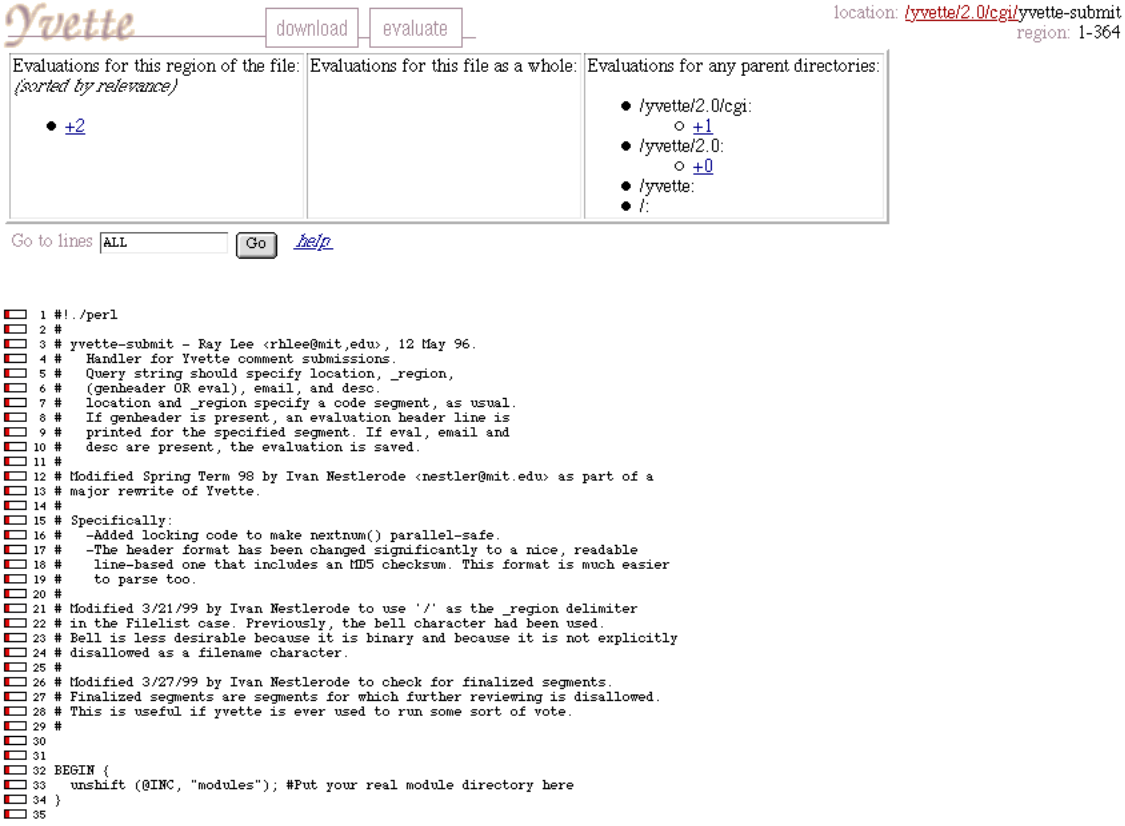

Figure 5: A typical evaluation. The small bars on the left of each source line are color-coded.

CHAPTER 4 *The Sample Application: Yenta*

4.1 Introduction

This chapter describes the *sample application*, named *Yenta*, that has been developed as a part of this research. The prior chapters are essential background for this discussion. Chapter 5 will evaluate the architecture and this sample application.

In this chapter, we shall describe:

- The purpose of the application—what problem does Yenta solve? Section 4.2
- Some sample scenarios—why might Yenta be useful? Section 4.3
- Yenta's affordances—what can users do with it? Section 4.4
- Political considerations—why this application in particular? Section 4.5

We shall then turn our attention to details of Yenta's implementation, and address:

- Yenta's implementation languages and internal organization Section 4.6
- How Yenta determines its user's interests Section 4.7
- How Yenta's security works Section 4.8

4.2 Yenta's purpose

Yenta has two primary purposes

- To serve as a distributed matchmaking system that can introduce users to each other, or form coalitions and discussion groups into which users may send messages to groups of others who share their interests (see Section 4.4). *Matchmaking*
- To raise public awareness for the political ideas about trustworthiness and protection of personal privacy advanced elsewhere in this thesis (see Chapter 1 and Section 4.5). *Getting the word out*

4.3 Sample scenarios

Before we examine exactly what Yenta can do, let us consider some sample scenarios.

> **You write software,** and you've having trouble with a particular tool. Somebody else just down the hall is using the same tool as part of what they're doing. But even though both of you talk every day, neither of you knows this—after all, this tool is just a little part of your job, and you don't tell everybody you meet about

every single thing you do all day. Yenta can tell you about this shared interest.

You're a technical recruiter. You'd like to find companies looking for people to hire, and people who are looking to be hired for your existing clients. They need privacy and anonymity, so the people they're working for now don't know they're looking. You need to be able to show them a good reputation, backed up by satisfied clients. Yenta is private and secure, *and* has a reputation system. Everybody's happy.

You're a doctor doing some research on an rare condition. Another doctor is doing the same sorts of research, but you don't know about each other. Maybe you're an academic, but you don't have enough to publish yet. Or perhaps you're a clinician, and don't realize that you're looking at a small part of a much bigger public-health problem. Yenta can help bring the two of you together, along with others who are studying the same problem.

You have an unusual interest, but you can't find anyone else who seems to share it. Maybe it's something embarrassing, that most people don't want to talk about publicly, so doing a web search hasn't turned up much. Yenta can help find others who share the interest, even if they don't publish about it. And it can keep the interest private, to only those who trust each other.

What do these scenarios all have in common? Users who may or may not know each other, but who do *not* know that they share an interest in something. Also, some of them depend on the existence of the reputation system, or upon the pseudonymous nature of how Yenta users are identified to each other.

4.4 Affordances

Let us now turn to Yenta's *affordances*, meaning exactly what functionality is made available to its users. This description is mostly from a user's standpoint—here, we describe more about what the user finds available in Yenta's set of possible actions, and less about how Yenta manages to do them.

4.4.1 User interface

Yenta communicates with its user by sending HTML to a particular network port, and instructing its user to connect to that port with a web browser. With the exception of the very first message from Yenta, in which it tells the user what URL to use, Yenta uses the user's web browser exclusively for its interactions. This has several major advantages:

Portability

- Supporting a graphical user interface is a tremendous amount of work, and is generally extremely non-portable across different types of computers. HTML, however, is extremely portable, provided that a lowest-common-denominator subset—essentially, that which has been approved by various standards bodies—is used. There are web browsers available for virtually all general-purpose computers in the world.

Familiarity

- By using HTML, Yenta can present its interface using a paradigm already well-known by millions of potential users.

Configurability

- If the user disagrees with some aspects of the UI—in issues such as font size, screen background, and so forth—it is generally possible to use the browser to change these, without having to support it directly in Yenta.

Security

- Yenta requires a high-security path from the user to Yenta itself, so that the user may type his or her passphrase without inordinate chance of it being eavesdropped. Common browsers support high-strength (128-bit session key) SSL connections, and Yenta uses cryptography exclusively when communicating with its user. We will have more to say about this in Section 4.8.

4.4.2 Yenta runs forever

Once Yenta has been started, it effectively runs forever. It disconnects from the controlling shell, and becomes a background process. While the user *can* manually shut

down Yenta from its user interface, this is discouraged, since it prevents other Yentas from communicating with the user's Yenta when the user is not attending it. This, in turn, means that Yenta will not perform as well as it could—it will miss opportunities for clustering and for passing or receiving messages. (Future versions of Yenta may not require permanent network connections, and will be more suitable from intermittently-connected machines, such as the dial-up connections employed by most home computer users.)

Checkpointing

Yenta checkpoints its state to disk periodically, and when it is shut down. This means that a machine crash can only lose a small amount of data; how often these snapshots occur, and thus the maximum amount of unsaved state that might exist, is configurable. For details about how this data is saved, and what precautions are taken to ensure both robustness and privacy, see Section 4.8.

4.4.3 Handles

Users of Yenta are identified by two types of names. The Yenta-ID was described in Section 3.4.1, and is essentially a 160-bit random number. This number is the fundamental way in which Yentas identify themselves to each other, and is both unspoofable and unique, as described previously.

YID's are precise, but cumbersome

A Yenta-ID is an unfriendly way to name entities which *people* must interact with—people are notoriously bad at remembering random 160-bit strings; they are very difficult to type; and they are more unique than is required almost all of the time. Furthermore, users generally prefer some degree of personalization of their online identities, and being able to choose their own name is a fundamental aspect of this.

Handles are nicknames chosen by users

Hence, Yenta also makes available a *handle*, which each user may set as he or she pleases. Handles are *not* guaranteed to be unique across any particular set of Yentas—indeed, since it is assumed that no Yenta knows of all other Yentas in the world, this seems impossible on its face. Handles provide a convenient shorthand when a user must refer to a particular other Yenta, and offer some degree of a chosen identity.

Local nicknames for others

Because handles are not guaranteed unique, users may also examine the Yenta-ID for a particular Yenta they communicate with, to avoid ambiguity. In addition, Yenta supports the ability to make *local nicknames* for any other Yenta's handle. This means that, if the user Sally is talking to some other user whose handle is Joe, but finds that she does not want to use that handle—either because she already knows two Joes, or because she simply doesn't like the name—she may instruct her Yenta to refer to the Yenta known elsewhere as Joe by some other name, such as Fred. This causes no confusion to other Yentas, which only refer to each other by YID anyway, and is invisible to everyone but Sally, unless she happens to mention her local, private nickname for Joe to anyone else.

4.4.4 Determining user interests

When Yenta first starts up, and periodically afterwards, it determines what the user is actually interested in. Without this determination, Yenta is useless—it would have no basis for which clusters to join, what introductions to make, and so forth.

Documents

Yenta uses a *collection of documents* to determine what a user is interested in. A single document is generally either a single file—if the file consists of plain text—or a single email message—if the file consists of several email messages grouped into a single file, as is popular with many mail-handling tools. Yenta can automatically determine, by analyzing the contents of the file, what sort of file it is, and whether it consists of a single document or several. The internal representation of a document is described in Section 4.7.

Scanning a tree

When Yenta starts up for the very first time, it asks the user for the root of a file tree. It then walks every file in that tree, rejecting those that appear to be binary files, and also rejecting portions of those files that appear uninteresting—email signature lines, the

stereotyped wording of header fields in email, HTML tags, PGP signatures, and so forth. It then *clusters* the resulting documents, as described in Section 4.7.

Single files

Once this initial clustering has taken place, users also have the option of directing Yenta's attention to a particular single file. This single file can be used to express a particular interest, perhaps obtained by the user importing a single document from elsewhere, and telling Yenta to give it disproportionate weight. In part, this makes it somewhat easier for users to express an interest in a particular subject so that they may find a group of experts.

Rescanning periodically

In addition, Yenta can be told to periodically resurvey the files it has already scanned. This allows it to pick up new interests as files are modified—files of email are typical for this. Interests from documents (entire files, or particular email messages) which are older than a user-settable threshold can be dropped, so that if the user loses interest in a topic, Yenta will stop trying to cluster based on it.

Giving Yenta feedback

Once Yenta has determined the user's interests, and at any time afterward that the user chooses, the user can survey the listing of accumulated interests and tell Yenta which ones are actually useful, and which ones are not. This has two important advantages. First, interests which were incorrectly determined by Yenta—such as a set of documents which contain some stereotyped text in each one, and which hence were clustered together—can be rejected. Second, not everything Yenta might find is equally important to the user. A common example is that of meetings: Most users working in white-collar environments in which meetings are scheduled by email will end up with a cluster containing words from messages such as *room, date, time, schedule, meeting*, and so on. For most people, just because someone else has meetings—on any topic—is no reason to suggest an introduction.

An example from Yenta's user interface of a set of interests is presented in Figure 6.

4.4.5 Messaging

Once Yenta has determined its user's interests, it engages in the clustering algorithm described in Chapter 2 to find other Yentas which share one or more of its user's interests. As soon as Yenta finds itself some clusters of others, it allows the user to send messages. These messages may be of two types:

- *One-to-one.* In this case, the user sends a message to a single other Yenta, which receives it and (usually) displays it to its user.
- *One-to-cluster.* In this case, the user sends a message to all the other Yentas in one of the clusters of which this Yenta is a member.

Examples from the Yenta UI may be found in Figure 7, Figure 8, and Figure 9. The implementation of how message-passing works is described in Chapter 2.

Grouping and filtering

Yenta users may group their messages by who has sent them, when they arrived, and so forth; the functionality resembles that of a typical mail-reading program. In addition, they may establish *filters*, which control which messages from other Yentas will be shown. These filters can screen out messages which do (or do not) contain certain regular expressions in their contents. In addition, rules can be written which use the attestation system (see below) to determine whether or not to present a message based on the reputation of its sender. This allows users to avoid seeing *spam* without ever seeing even the very first message from the sender—by instructing Yenta to ignore messages which do not meet some reputation criteria, spammers who fail to acquire a good-enough reputation become invisible to the given user. Of course, care must be taken in writing such rules, lest most other people be inadvertently lumped into the group of potential spammers.

4.4.6 Introductions

If one user's Yenta determines that some other Yenta seems unusually close in interests to one of its users clusters—better the characteristics match within a user-settable threshold—it can suggest an *introduction.* This suggestion takes the form of an automatically-generated message to both Yentas. The Yenta suggesting the introduction sends a message to the other Yenta saying, in effect, *I think we should be introduced,* and also, in effect, sends its user a message saying, *I think you should introduce yourself to this other user.* Users are free to accept or ignore such introductory messages, and may configure Yenta to increase or decrease the approximate frequency of their occurrence.

Introductions serve the important purpose of getting users together who do not otherwise know of each other's existence. After all, if user A sends a message to B, then A must have known about B first. Similarly, if user A never sends any messages, even to a whole cluster, then A is effectively invisible to everyone else in the cluster. Introductions serve as a way to suggest to such *lurkers* that they interact with particular other individuals.

4.4.7 Reputations

Chapter 2.11 described the basic features of the *attestation system,* in which users may create strings of text describing themselves, and others may cryptographically sign these strings. Yenta supports the creation, display, and signing of attestations, and users may use these attestations to filter incoming messages based on who has signed the attestation or which strings appear in an attestation.

The actual things that users say about themselves via this reputation system constitute a set of social mores. The final development of this set is unknown; it is very often the case that small initial perturbations can lead to large eventual changes in what are considered common customs, idioms, and the like [20][33][49][59][60][116]. The study of how Yenta's users actually use the reputation system could be very fruitful from a sociological standpoint.

See Figure 10 for an example from Yenta's UI of how attestations are seen by the user.

4.4.8 Bookmarks

It is often convenient to be able to mark a spot in the user interface with a bookmark, similarly to the way that one can bookmark a page at a static website. However, Yenta makes this more complicated than it might appear, because there may be more than one Yenta—each belonging to a different user—running on the same computer at the same time. As explained in Chapter 2.12, this means that each Yenta must use a different network port to communicate with its user—but browser bookmarking systems include the port as part of the URL. Consider what happens when the user drops a bookmark on some page of a running Yenta. When that Yenta is later restarted—after a machine crash, or because the user shut it down to start running a newer version—there is no guarantee that it will acquire the same port. Any browser bookmarks will therefore be invalidated, pointing either at the wrong Yenta, or no Yenta at all.

To avoid this, Yenta has its own, internal bookmarks, which may point at any page served by the user interface. Users can add or delete bookmarks, and may sort them either alphabetically by page title, or chronologically by when they dropped them. Since these bookmarks are kept internally by Yenta, the details of which port Yenta happens to be currently using for its HTTP server are irrelevant.

4.4.9 News

Yenta occasionally has something to say to the user that is unrelated to anything the user has done recently, and is also not an incoming message. For example, someone may have recently signed one of the user's attestations, and their Yenta has just connected and passed it along. Or Yenta may have decided to rescan the user's documents, based on instructions to do so periodically, and may wish to inform the user that this has taken place.

In these cases, Yenta makes available a page of *news*. Each item on this page is a brief description of some event that has taken place. Users may review the items, and then tell Yenta to either keep each one or discard it. See Figure 11.

4.4.10 Help

Yenta contains a large number of pages documenting its operation. Users may select such pages at any time. The help system understands which page the user was just viewing, and can sometimes offer a specific help topic that would be relevant to the page the user just came from. However, users can always see all available help topics at any time. See Figure 12.

4.4.11 Configuration

Users may tune certain parameters in Yenta to make it more to their liking. For example, certain thresholds, or the details of what constitutes a file which Yenta should ignore during scanning, may not be correct for all users in all environments. Yenta allows users to adjust the values of these parameters. See Figure 13.

4.4.12 Other operations

A few infrequently-used operations are gathered together on a single page; see Figure 14. These include, for example, allowing the user to change his or her passphrase. In addition, this is how the user can cleanly shut down Yenta by hand, for example if the host machine is about to be taken down. Failure to shut Yenta down in this circumstance means that any changes to its state—such as incoming messages—since the last automatic checkpoint will be lost. The very last page presented by Yenta's user interface in this case is shown in Figure 15.

4.5 Politics

There are several reasons why this particular sample application was chosen to illustrate the political goals of this research.

First, by basing its assessment of user interests on users' own electronic mail, Yenta starts with a set of data that is already quite likely to be considered private by its users.

Because Yenta thus deals with private information so heavily, a solution which does not make the usual compromises—weak or no encryption, and a central server which collects everything—was imperative. Without such a solution, user acceptance of Yenta would be slight.

There is a great pent-up demand for the problem that Yenta attempts to solve—namely, matchmaking people and finding interest groups. For example, at one time, Yenta was nothing but a set of proposals, some research papers on simulation results, and a vaporware description of what its implementation would probably look like. Nonetheless, the author received (and continues to receive) several *hundred* messages every year asking for a copy of the application. Even though, at the beginning, deployment of the application was stated to be quite some time away, response to this otherwise-unadvertised potential application was impressive.

This combination of private information, an architectural solution, and great user demand means that the Yenta application can itself be an exemplar, which by its very existence advertises that it is possible to offer the service that it does *without* the traditional compromises that users have come to expect. In addition, the matchmaking that Yenta does—allowing people to communicate more easily—is itself a social good, irrespective of its intended effect on later applications designed by others.

Of course, this stance does not come without a price. For example, Yenta's use of strong cryptography means that the application itself, having been written inside the United States, may not legally be exported outside the United States and Canada [50][87]. This complicates Yenta's deployment—it requires that the distribution site run a script that checks the location of the user requesting the download, and ensures that the user at least professes not to be interested in violating US export-control reg-

ulations. Furthermore, it means that Yenta may not be mirrored by other sites, unless they arrange to do the same.

4.6 Implementation details

Yenta is actually implemented as four major subsystems:

- The cryptographic engine, *SSLeay* [186].
- The document feature extractor, *Savant [146]*.
- The Scheme interpreter, *SCM [89]*.
- The main functionality of the application.

Safety via code reuse

In general, the strategy applied was to reuse, not rewrite, those components that Yenta required and that were already freely available. Not only is this the expedient course of action, in the case of Yenta's cryptographic elements, it is also the *safest*—cryptographic software required careful review, because even a good algorithm and design can be ruined by incorrect implementation. Hence, Yenta does *not* use its own low-level cryptographic infrastructure—it uses code that others have carefully reviewed as much as it can. Local modifications to such code, while required to achieve the functionality Yenta requires, are made carefully.

The resulting system is composed of approximately 240,000 lines of C, and 15,000 lines of Scheme.

4.6.1 The C code

The first three of the subsystems above are implemented in C, and come from outside the Yenta project per se. SSLeay [186], which is also used in popular versions of the Apache [7] web server, was written in Australia over a span of many years, and has been vetted by many developers who use it in their own applications. Savant started out in life as the original Yenta document comparison engine. This engine originally used the SMART [188] document comparison engine from Cornell, and later was completely rewritten locally to include only the functionality required by Yenta—SMART was too large, too buggy, and did not really do what we needed to do. This code then became the basis for the document indexing engine—Savant—which itself also a part of the Remembrance Agent [146], and was then handed back to Yenta—in short, this code has been getting shared and rewritten between two research projects for years. Finally, SCM [89] was written by a guest of the MIT Artificial Intelligence laboratory, again over a period of years.

Code reuse is hard

We have made our own modifications to all three of these packages, rewriting or extending each one by 10-20% (in terms of lines of code) to make them exactly what Yenta requires. While Yenta's development would have been impossibly complex if all of these packages were to have been written from scratch, its requirements are sufficiently unusual that nothing was quite correct out-of-the-box. Savant, for example, required extensive changes so that it did *not* assume it could touch the disk whenever it wanted (as the Remembrance Agent assumes), and also had little support for the document-clustering that Yenta performs. SCM required major modifications to enable reliable networking, to hook it into the SSLeay crypto API, to not make assumptions about the environment in which it would be run, and to enable shipping a single binary, consistent of the entire application, on a wide variety of machine architectures.

Portability

Yenta is designed to be easy to port. One of the modifications made to all three of these packages was to place each of them under the GNU autoconf/automake system [109], which allows extremely fast configuration of a C-based system on almost all UNIX hosts. This means that someone who wishes to build Yenta from scratch, in

many cases, need type only *./configure; make* to build the entire C side of the package.

4.6.2 The Scheme code

Most of the unique functionality of Yenta is written in Scheme. This was done for several reasons:

- Scheme, like many Lisp-based languages, solves many traditional problems such as garbage-collection and exception-handling in a clean, elegant way. This is a much larger benefit than it first appears—in a C program, every line of code is a potential coredump, segmentation violation, or memory leak. Yenta must be *robust* if its users are to take full advantage of it. One of the easiest ways to ensure this robustness is to write in a language which can correctly handle these details for the programmer.
- Scheme is not only safe against *crashes*, but confers substantial safety against *malicious attack*. Approximately half of all crack attempts against operating systems and applications which are written in C consist of *buffer-overrun attacks*, in which a deliberately-too-large string is sent to some piece of code which fails to correctly check the size of the buffer for which the data is destined. In the most commonly-used environments, such as attack is over used to overwrite the program control stack and force the application to execute arbitrary code from elsewhere that has been embedded in the data. Scheme cannot fall victim to such an attack, because *all* such data structures are automatically checked by the interpreter for safety before execution.
- Scheme code is quite compact. An informal estimate of Yenta's code, and of similar other projects, indicates that 1 line of Scheme code typically takes the place of 10 or more lines of C code, when integrated over a large project.
- Part of Yenta's purpose is pedagogical—it exists to show how to write distributed, privacy-preserving applications. By writing a large portion of it in Scheme, its underlying principles can be more easily revealed without being hidden under a huge amount of otherwise necessary but verbose code.
- The SCM implementation runs on a very large selection of platforms, including not only UNIX, but MacOS, MSDOS, Amiga, and others. This means that code written in Scheme is inherently quite portable, and simplifies the task of making Yenta run on a large variety of platforms.
- Despite the fact that Scheme is an interpreted language, the SCM implementation used in Yenta has proven itself to be very fast. We have not observed that the user need wait for Yenta, at any point, because of any inefficiencies introduced via the user of an interpreted language.

The actual Scheme code of Yenta is roughly divided into several subsystems:

- *The task scheduler.* Yenta internally runs a dozen or more individual tasks. Each task handles one I/O stream, such as communicating with a single other Yenta, or with the user's web browser. In addition, various tasks run autonomously at various times to checkpoint Yenta's state to disk, dump statistics to the statistics-collection server, rescan the user's files for new interests, and so forth. Each task is *non-preemptive*, due to the nature of the SCM implementation—it must explicitly yield to the next task—and there is substantial support implemented to make it easy to write tasks in this manner. Some tasks have higher *priorities* than others—for example, the user-interface task runs at very high priority, so the user is never left hanging, waiting for a page to load. This reassures the user that Yenta is, indeed, still functioning. Finally, *tasks which get errors are handled*—this includes saving a backtrace of the task for debugging and sending it to the debugging-log server for later analysis by the implementors. Typically, a single dead task only momentarily interrupts communication with a single other Yenta, or disrupts a single browser page fetch, and does not permanently cripple the running Yenta. (Yenta tasks encounter errors only very rarely, and their incidence decreases as Yenta's code becomes more

completely debugged. A system with zero bugs, of course, could be expected to never have a task get an error—but even though Yenta is presumed not to be at that point yet, handling errors in this way makes it much less likely that Yenta fail completely due to an error in one part of itself. This makes Yenta substantially more robust than much existing software.)

- *The user interface.* This code understands how to speak HTTP to a browser, including using SSLeay to encrypt the connection, and can produce correct HTML for each page to be shown to the user. Pages are written for the most part in plain HTML, but they may *call out* to Scheme code to generate part of the page—thus, for example, a page may have a constant paragraph of text, and a dynamically-generated table, whose contents are based on Yenta's current state.
- *The InterYenta protocol engine.* This manages communications with other instances of Yenta running elsewhere.
- *Interest-finding and clustering.* Yenta must keep track of the user's interests, and must both communicate those interests to other Yentas, and allow the user to tweak them.
- *Major affordances.* Yenta has a large number of various capabilities—message origination and reception, attestation management, and so forth. Most of these affordances use the code described in previous bullets as infrastructure, and is therefore relatively compact and easy to implement once the infrastructure is in place.

4.6.3 Dumping

Yenta is built in two pieces. First, all of the C code is compiled and linked, yielding a highly-customized version of SCM that also incorporates the Savant and SSLeay libraries. Then, the binary is run, and all of the Scheme code and web pages are loaded into the Scheme heap. Once they have been loaded, Yenta is *dumped*. This process creates a single file which is a snapshot of the original C code, the contents of the heap, and a continuation which is the locus of control when the binary is restarted.

Yenta is a single binary file

This procedure means that Yenta may be shipped as a single binary, with no ancillary files of any sort. Users who download the binary may simply run it as-is, with no compilation or configuration steps. Making this process trivial was a high priority in Yenta's design, since even the vast majority of UNIX users would find it either inconvenient or impossible to actually compile an application from source. A very small percentage of those who *might* run Yenta actually *would* if they had to build it from scratch. Of course, since Yenta's source distribution is public (subject to export restrictions), anyone who wishes to build Yenta, either because they do not trust the binaries, or because they need a binary for some machine not already available, is free to do so.

4.6.4 Architectures

Because ease of installation was a design priority, Yenta is distributed with precompiled binaries for popular UNIX platforms. As of this writing, this includes Red Hat Linux 5.1, NetBSD 1.3.2, HPUX 9 and 10, SGI Irix 6.2, and Alpha OSF1. It is quite likely that Yenta will compile with no work on many other architectures, but these were the only ones routinely available to the author.

4.7 Determining user interests

Given a collection of documents, as detailed in Section 4.4.4, how does Yenta actually determine the user's interests?

4.7.1 Producing word vectors

The first step consists of turning each document into a weighted vector of keywords. Each keyword corresponds to some word that appears in the original document, with certain modifications [146]:

- Very common words (stopwords) are removed. *Toss stopwords*
- Anything matching an *exclusion* regular expression is removed. This gets rid of HTML markup, PGP signature blocks, base64-encoded MIME documents, mes- *Toss machine-generated phrases*

sage header field keywords (e.g., anything to the left of the colon in an RFC822 email header field), and a large number of similar elements.

Stem

- The remaining words are *stemmed* [133] to remove suffixes. This causes words which have the same root, but are used as different parts of speech, to be more likely to match. Note that this step of the algorithm is English-specific; if Yenta was ever ported to some other language, the logic of this stemmer would have to be modified.

Weight and vectorize

The number of times each resulting word occurs in each document is then counted, and the result normalized by the total length of the document. This ensures that long documents do not disproportionally weight the results. The end result of this process is a *word vector*, which details, for each document, which interesting words occur in it.

4.7.2 Clustering

The second step of the process produces *clusters* of documents which appear to be talking about similar topics. Each one of the clusters formed is potentially one of the user's *interests*, and is what is referred to more generally as a *characteristic* in Section 2.6.

The algorithm which forms the clusters operates as follows. We pick a random starting vector, *V*, and then pick a second vector, *W*. We dot the two vectors together, which determines the similarity of one vector to another. If they match within a threshold, both vectors form the start of some cluster *C*. If not, we let *W* also be the start of a new cluster, and pick a third vector, *X*, dotting that against the two vectors we already have. Any close match joins its cluster; bad matches form their own clusters.

After we have generated a few clusters, we stop attempting to generate more, and simply dot the remaining vectors against vectors already in clusters. (For efficiency, we maintain a moving-average representation of each cluster's centroid; this means that testing a vector against a cluster requires dotting it against only one average vector, and not against each vector in the cluster.)

When this terminates, we are left with a collection of clusters, and a collection of vectors which were not similar enough to any already-existing cluster to wind up in one. The next step is to investigate the fitness of each cluster—after all, the moving average centroid of any given cluster might have left behind the first few vectors to have been added. This can happen if we are unlucky in our choice of initial vector, and the centroid shifts a large amount due to later additions.

Thus, we *prune* already-existing clusters by dotting each vector already in a given cluster against that cluster's centroid vector. Vectors which are no longer close enough are discarded again.

We are now left with some pruned clusters and a pile of extra vectors. This latter pile is made up of vectors which never made it into a cluster in the first place, plus vectors that have been discarded from existing clusters. It is possible that some of these vectors are sufficiently alike that they could form a cluster of their own, so we start the clustering process again, using this pile of discards—one of the vectors we start with may form the seed of a new cluster. After the initial cluster-formation step, we check each vector in the discard pile against *all* clusters we have generated, and keep any good matches.

This algorithm iterates, controlled by thresholds at various points, until some proportion of vectors are in clusters, and enough iterations have run. We are left with clusters that have empirically-reasonable variance in terms of the vectors they include, and a pile of leftover vectors.

The algorithm actually runs the forward (clustering) direction and the reverse (pruning) direction in parallel. This is analogous to the way bone is formed—via cells called osteoblasts—and destroyed—via osteoclasts. Bone is piezoelectric, and generates an electrostatic field when under mechanical stress. Osteoclasts are constantly tearing down bone, whereas osteoblasts produce more bone wherever this is a large electrostatic field. Hence, bone preferentially builds up wherever the stress is highest—hence reducing the stress again—without building up in places where it is not needed. Yenta's document-clustering algorithm constantly tries to build up a cluster by adding any vector which is close to that cluster's centroid, while it simultaneously tries to tear down the cluster by removing any vector newly deemed unfit to remain.

The initial vectorizing algorithm, which converts documents to vectors of keywords, runs in time and space that is approximately linear in the number of words in all documents. The clustering algorithm is slightly more complicated. The forward direction runs in approximately linear time, due to its use of the moving-centroid approach. The reverse direction runs in approximately $O(n^2)$ time, since the total number of times any given vector might be chosen to compare against the centroid depends on the size of the cluster and how long this cluster has been around. However, since the number of clusters—generally under a hundred, and often under twenty—is much smaller than the typical number of documents—which typically number in the thousands or more—the overall behavior of the clustering algorithm is typically close to linear.

The algorithm chosen here was simply generated ad-hoc. We shall have more to say about its performance in Chapter 5, but the overall lesson is that it seems to work well enough. Since Yenta makes no particular claims either to advance the state of information-retrieval research, nor of optimality across any particular dimension of document comparison, this is acceptable.

4.8 Security considerations

We spoke at length about the security of the general architecture in Chapter 3. Here, we shall speak about a few wrinkles that Yenta introduces.

4.8.1 Encrypting connections

Connections between Yentas, and connections from Yenta to the user's browser, are always encrypted. This is accomplished by running SSL [186] between each pair of communicating agents, using Diffie-Hellman key exchange for *perfect forward secrecy*—session keys are discarded at the end of the connection—and self-signed certificates to complicate *man-in-the-middle* attacks.

Note that these self-signed certificates make it difficult to do a man-in-the-middle attack only between two Yentas (or a Yenta and a browser) that have previously communicated. They are worthless if a man in the middle can be in the middle from the very start of the conversation, since there is no certifying authority, nor a web of trust, available to validate the cert. On the other hand, since we are in the case that we have never talked to the Yenta at the far end of the connection anyway, we might as well treat the man in the middle as just some *other* unknown Yenta we have never spoken with. The man in the middle can keep both ends from knowing the true YID of the endpoints, but it cannot otherwise cause much trouble—for example, attestations are signed by other Yentas, not by the Yenta belonging to the user the attestation refers to. Indeed, were someone to set up a man-in-the-middle Yenta that successfully passes data in both directions to two other Yentas, the largest apparent problem surfaces if the middle Yenta vanishes—at that point, neither endpoint knows how to talk to the other.

4.8.2 Protecting persistent state

Yenta must save persistent state to disk. If it did not do this, it could not survive the crash of either Yenta or the host computer. There are two cases here: the user's characteristics, and everything else.

Characteristics

First, we have the user's characteristics, which were derived from the user's file and email. These are stored unencrypted, for two reasons:

- The characteristics were originally derived from reading messages which arrived over cleartext channels, and are stored on the disk in the clear. The original representation of this data (files and email) is far more comprehensible to humans than the vectorized, stopworded, stemmed representation left on disk—hence, leaving this data around on disk, assuming it is at least protected against other readers using filesystem protection bits, is no more of a privacy exposure than what the user was already doing.
- The Savant library is unprepared for dealing with encrypted data. If we did not also have the case detailed in the above bullet, it would be worth fixing this. As it is, however, such effort would not improve Yenta's privacy.

Keys, conversations, ...

Even though the user's characteristics were derived from the user's mail—presumed to already be sitting around on disk in the clear—the stored conversations in which the user has participated were not formerly stored in the clear, and were carefully transmitted between agents using encrypted protocols. We should not presume to expose them once they have been stored on disk. Even worse, the user's private key—the very basis of his or her identity—is in the same file. Exposing this would be a disaster, since it could allow anyone to both eavesdrop and impersonate the user.

The strategy used is to encrypt the data directly to disk, using IDEA in cipher-block-chaining (CBC) mode. It uses *ePTOBs*, aka *encrypting Scheme port objects*, which act like normal Scheme ports, but encrypt or decrypt along the way—they use SSLeay for their underlying implementation. The question then is, how does Yenta store the key so the data may be decrypted later?

What it does is to write out a small preamble, which consists of some bootstrapping data, and then the main data, which consists of the encrypted state. Both of these are written to the same file on disk.

Yenta's actual *persistent state* is a variable-length string of bytes, called ***D***. [We do not compute a MAC of ***D***; perhaps we should if we can. This would provide some protection against an attack that changes bit(s) of ciphertext (hence trashing the plaintext), but it would require somehow either precomputing a checksum, or computing one on the fly as data is written out. Both are somewhat inconvenient.]

When Yenta first starts up, it asks the user for a *passphrase, P.* This passphrase does not change unless the user manually changes it. Yenta immediately computes the SHA-1 hash of the passphrase, P_{SHA}, and throws away *P.*

Saving state

Each time Yenta needs to save state, it generates a new 128-bit *session key, K*, which is used for keying the cipher. It also generates a 64-bit *verifier, V.* Both of these are high-quality random numbers, drawn from the random pool. Finally, it generates an *encrypted version of the session key,* K_P, using the first 128 bits of P_{SHA} as the encryption key and IDEA as the cipher. (Since we're encrypting 128 bits of random data, we need neither any block-chaining, nor any IV.)

It then writes out the following data

- To the preamble (*key*) portion of the file, *in the clear*:
  - The *cleartext version of the browser cert*
  - The *encrypted version of the session key,* K_P.
- To the main (*data*) portion of the file, *encrypted on the fly* (via an ePTOB keyed by *K*, the *session key*):
  - Two copies of the *verifier, V,* one immediately after the other; we shall call this V_1V_2.

- The *persistent state, D.*

Because data is encrypted on the fly, before it hits the disk, what we have really written to the main data portion of the file is really $[V_1V_2]_K$ *and* D_K.

Restoring state

Yenta only reads its persistent state upon startup. The first thing it must do is to read the *cleartext version of the browser cert* from the keyfile. It requires this data so it can establish an SSL connection to the user's browser, without generating a brand-new certificate—doing so would require that the user walk through all the cert-validation menus in the browser for every Yenta startup.

Yenta then prompts the user for the passphrase, *P*, and computes P_{SHA}, as above.

It then reads the encrypted session key, K_P, from the preamble, and decrypts it, using the first 128 bits of P_{SHA} as the key. This regenerates the true session key, *K*.

Now that *K* is known, Yenta continues reading, now in the encrypted portion of the file, and reads the first 128 bits from it, which should be V_1V_2—the two concatenated copies of *V*. If V_1 does not match V_2, then *K* must be incorrect. For *K* to be incorrect, we must have incorrectly decrypted K_P, which implies that P_{SHA} is wrong. The only way this could happen is if the user mistyped the passphrase, so we prompt again, and repeat.

Assuming that the verifier matches, we now have a correct session key, so we supply that to the decrypting ePTOB and read the rest of the file, which converts D_K back to *D*.

Vulnerability analysis

What vulnerabilities might exist in this approach?

- Data is never left unencrypted anywhere on disk.
- We assume that IDEA-CBC is secure up to brute-force keysearch. Nonetheless, we assume that we do not want to gratuitously enable a known-plaintext attack. [The ePTOB itself also includes a 64-bit IV before the encrypted data; this helps to foil known-plaintext attacks on the first block. This would otherwise be a *very* simple attack, since the contents of the first block are nearly constant for all Yentas.]
- We assume good random numbers.
- We are *not* secure against an attack that can read the contents of Yenta's address space. (This is true of the entire design: anyone who can read the address space can suck out P_{SHA}, which is kept around indefinitely. This does not matter, though, because such an attack could suck out the RSA keypair which defines the basis of the user's identity—this is far worse, and is basically a complete compromise, allowing both eavesdropping and spoofing.)
- A weak passphrase is vulnerable to dictionary attack, which will allow decrypting the session key and thus allow access to the plaintext of the private key.
- It is possible that $[V_1V_2]_K$ could leak some information to a cryptanalyst. E.g., it is known that 4 bytes are repeated in the next block in a predictable place in the ciphertext (since we use an IV but not variable padding). This does not appear to be an actual vulnerability, since *V* is not known plaintext. (Hashing the second copy might help even so, or might only add a constant factor to the attack; not clear.)

It *appears*, as usual, that the primary vulnerabilities are (a) insecure process address space, and (b) the user picking a poor passphrase.

Full disks

There is one final consideration. What happens when the disk fills up?

Yenta tries to be relatively careful about the integrity of the saved statefile. After all, if this file is corrupted, the user's private key goes with it, and hence all of the user's identity and reputations (via attestations signed by other users) as well. This is an intolerable loss.

The most obvious defense is to write a temporary copy of the statefile, ensure that it is correct, and then atomically rename it over the old copy. This means that a crash in the middle of the write will not corrupt the existing statefile. But how do we know that the tempfile was, in fact, written correctly?

SCM does not signal any errors in any of its stream-writing functions, because it fails to check the return values of any of the underlying C calls. This means that, if the disk fills up, the Scheme procedures *write*, *display*, and related functions will merrily attempt to fill the disk to full and bursting, and will continue dumping data overboard even after the disk is full, all without signalling any errors. This is an unfortunate, but hard to fix, implementation issue.

Even if we check at the beginning and the end whether the disk is full (by writing a sacrificial file and seeing if we get the bytes back when we read it), consider what happens if the disk momentarily fills in the middle of saving state, then unfills. This could easily happen if something writes a tempfile at the wrong moment. In this case, SCM will silently throw away *n* bytes of intended output, while not detecting the failure. Even rereading the file may fail to detect it, if the dropped bytes were inside a string constant. One possible solution is to call *force-output* after every single character, then *stat* the file and see if its length has incremented, or, alternatively, to write and read a sacrificial file after each character of real output. Either of these approaches is (a) extremely difficult to implement (since we write output in larger chunks, and through an encrypting stream as well), and (b) horribly inefficient, probably slowing down checkpointing by at least two orders of magnitude if not more.

To avoid this, we run a verification function over the data written, every time it is written. This function does the work of reading and checking the contents of the preamble against the running Yenta (e.g., encryption protocol version, the browser cert and browser private key, etc.), and then computes the SHA-1 hash of the entire encrypted portion of the file, e.g., of the *data* portion in the discussion above. This is then compared with an identical hash, computed seconds earlier when the data was written to disk. If anything is wrong with the preamble or if the hashes do not match, then *something* is wrong with the data we just wrote; a single byte missing or even a single bit trashed will be evident.

In this case, we do *not* rename our obviously-corrupt tempfile over the last successfully-saved statefile. Instead, we delete it again, since it may be contributing to a disk-full condition and is bad in any event. In addition, we set a variable so the user interface knows that something is wrong, and can tell the user, who can presumably attempt to fix whatever is preventing us from successfully writing the statefile.

Note that this gives us no protection over having the statefile trashed *after* we have checkpointed. If Yenta is still running, the damage will be undone at the next checkpoint, since the old file will simply be thrown away unread. However, if Yenta was not running when the file was trashed, Yenta will simply fail to be able to correctly read the entire thing. (Chances are overwhelming that any corruption of the file will yield garbage after decryption that *read* will complain about, and Yenta will be unable to finish loading its variables.) In this case, the user will have no choice but to restore the file from backup. This is the expected case anyway if files are being trashed at random in the filesystem.

Note also that Yenta's support applications, which write plaintext statefiles and do not save state using encryption, do *not* do this checking. They save very little irrecoverable state in normal operation; the big exception is the statistics logger, which will simply have its data truncated, losing log entries that arrive while the disk is full and possibly leaving a corrupted last entry. This is not considered a serious problem. Fur-

thermore, this state is not being saved in a statefile at all, but is being explicitly written to a a separate logfile.

4.8.3 Random numbers

Yenta's security is dependent upon having good random numbers, since these numbers determine the quality of its cryptographic keys. On machine architectures which have */dev/random*, Yenta simply uses that—it is designed to be good enough for most cryptographic applications, and tries hard to collect random state from all over the machine.

Machines which lack */dev/random* instead prompt the user, the very first time Yenta starts up, to enter a large number of keystrokes, and Yenta measures the interarrival time of these keystrokes. This is the same technique (and partially the same code) used by PGP.

Yenta then maintains that random state by keeping a *random pool*, which is a collection of random bits. Everything that uses bits from the pool, such as generating a key, keeps track of the number of bits used, and Yenta runs several tasks at a variety of time intervals which attempt to regenerate randomness in the pool by running a variety of programs which sample many events happening on the system, as well as also using */dev/random*, if available. This random-pool is saved when Yenta checkpoints its state, so newly-started Yentas have randomness. As long as Yenta can continue to gather randomness data from the machine faster than it is consumed to generate, e.g., session keys, its cryptographic quality should remain high. (If Yenta cannot do this, it warns the user; this is considered an implementation error.)

4.9 Summary

In this chapter, we have described Yenta—the sample application which demonstrates how the underlying architecture can be used in a real system, and which is intended to raise public awareness of the techniques developed in this research and the rationale for their development. We have presented several sample scenarios to motivate why Yenta is useful, and then described the various affordances provided by Yenta to its users. These include automatic determination of interests, messaging into groups of users who share interests or to particular individuals, automatic introductions, and a reputation system. We then delved into Yenta's implementation, describing the general structure of the code, what the major pieces are, and how they fit together. Finally, we discussed those security considerations which are specific to Yenta itself and not necessarily to the general architecture.

Figure 6: A sampling of interests. Real users tend to have many more than shown here.

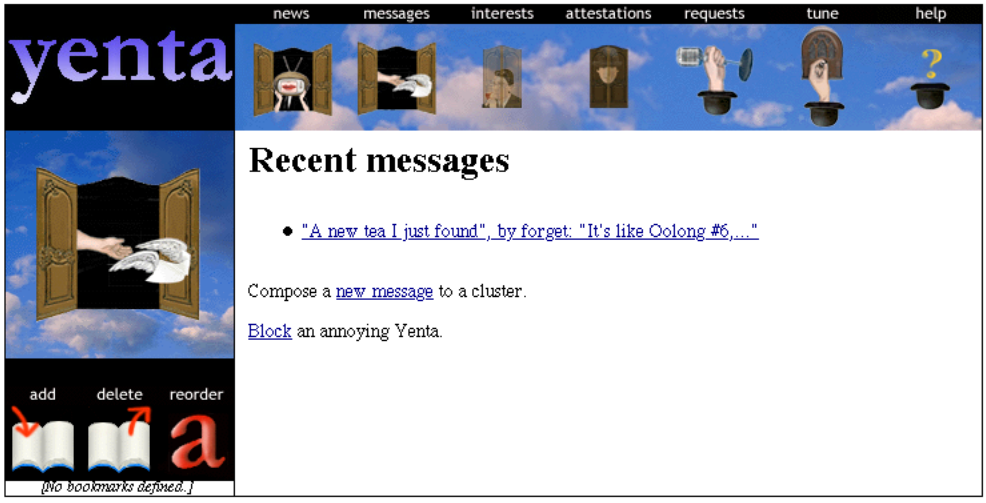

Figure 7: Recent messages received by this Yenta, and options for dealing with them.

Figure 8: A typical message, and how to reply.

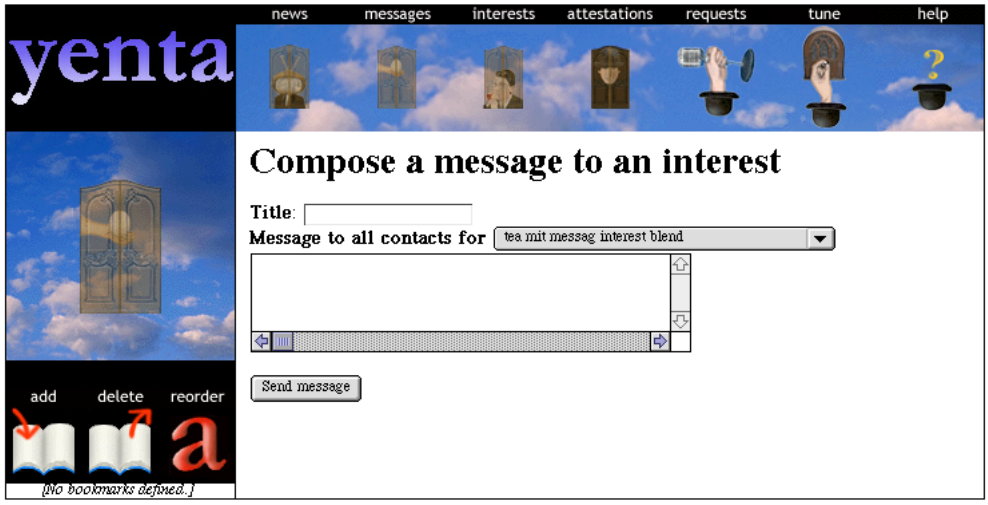

Figure 9: Replying to a message.

Figure 10: Manipulating attestations.

Figure 11: Recent news about this particular Yenta.

Figure 12: A sampling of the help.

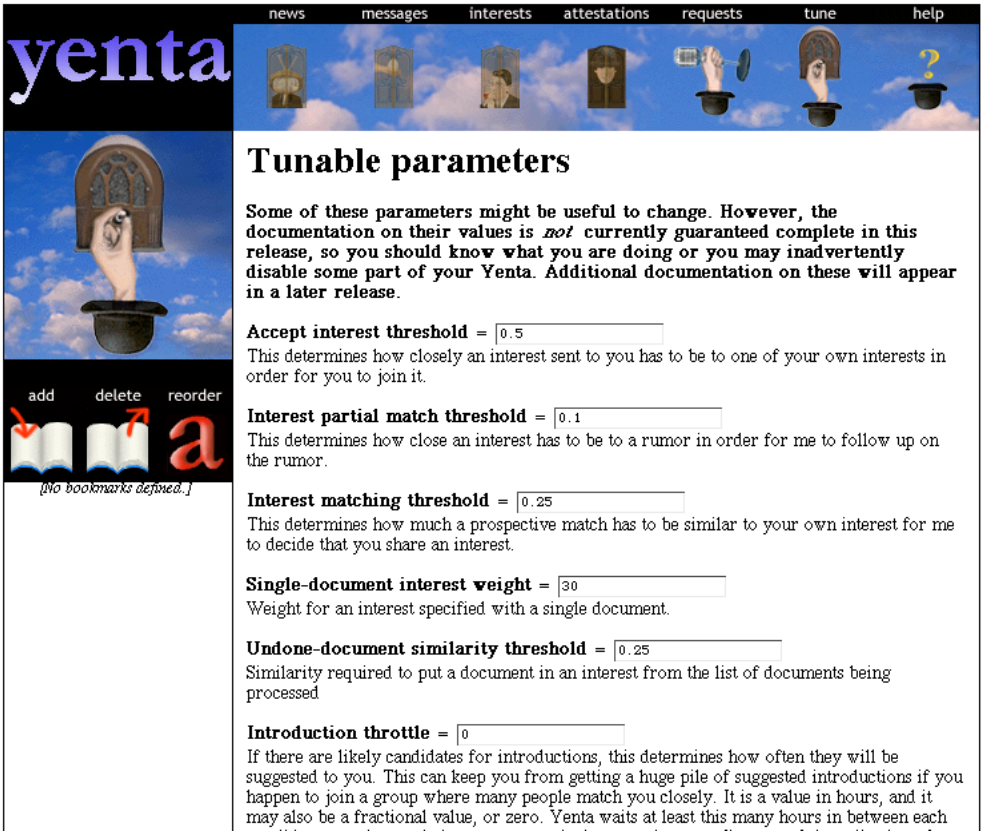

Figure 13: Adjusting internal parameters, for those who demand knobs..

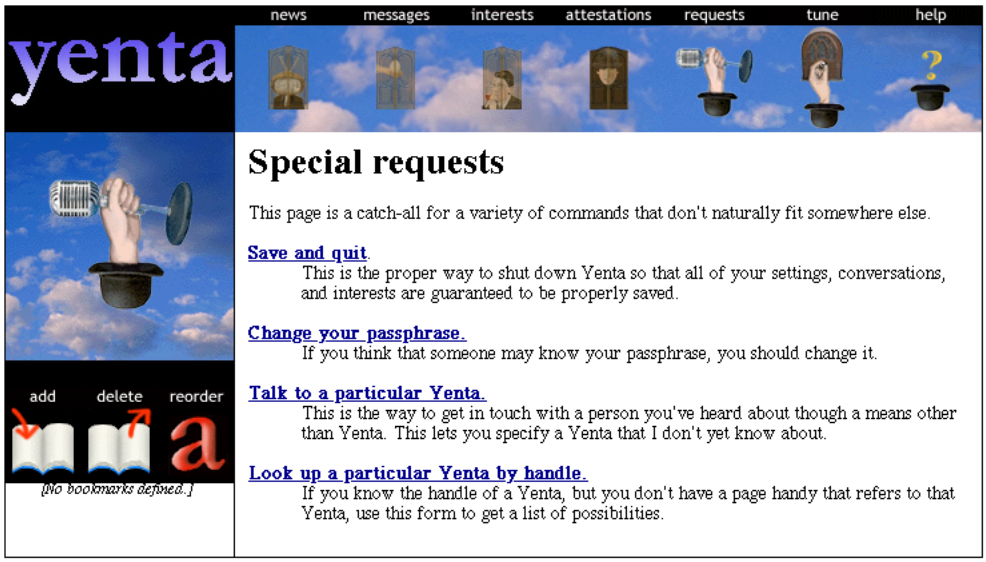

Figure 14: Some infrequently-used operations

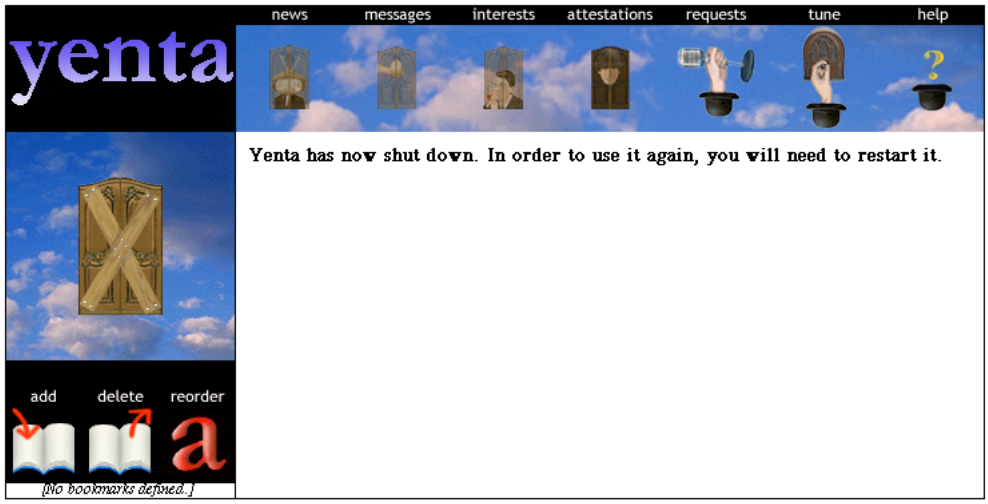

Figure 15: If Yenta is manually shut down, this is the last page it shows.

CHAPTER 5 *Evaluation*

5.1 Introduction

This research proposes an architecture with political ramifications, and a sample application that demonstrates how such an architecture can be used. What are suitable metrics for evaluating it? The space of possible evaluation strategies, and the questions that could potentially be asked, is quite large. In this section, we shall whittle the problem down a bit. Later sections will cover:

- Simulation results of Yenta's network clustering algorithm — Section 5.2
- How can we collect data from running Yentas? — Section 5.3
- What data is currently collected? — Section 5.4
- What are some of the questions we can answer? — Section 5.5
- How can we evaluate Yenta's security? — Section 5.6
- A risk analysis of the architecture and the fielded system — Section 5.7
- Other applications suited to this architecture — Section 5.8
- Ideas on motivating businesses to use this technology — Section 5.9
- Future work — Section 5.10

First off, we aim to show that one can design an architecture which can *protect*, yet still *use*, personal information when implementing applications with certain characteristics. The architecture, and the types of applications for which it is suitable, was described in Chapter 2, and the particular application used to investigate the architecture was described in Chapter 4. We have claimed that this architecture is an advance over traditional methods of handling this information for the same types of applications.

The eventual goal of this research is to encourage system designers to change the way they design systems—in particular, to start from a social agenda and design forward from that, rather than ignoring such an agenda or assuming that it will hinder building systems that people can use for useful tasks. This is an essentially political motivation which attempts to give users systems that are more robust against failures and more likely to protect their rights. Actually observing such a change, however, involves a long timescale—we would be dependent upon finding some system which *was* going to be designed in a centralized, non-privacy-preserving fashion, but which is *now*

going to be designed differently, because of the research described here. Such designs take time, and often happen in with little public disclosure.

As discussed here, therefore, we are *not* aiming to show that the architecture has already made an impact, nor are we aiming to show that the sample application—Yenta—has or will be used sufficiently for the its political impact to be felt. That is beyond the scope of this dissertation—but certainly not beyond the scope of the political agenda that motivates the work. Indeed, one of the hopes in this research is that it will *contribute to the discourse* surrounding the design and implementation of such systems.

Therefore, evaluation focuses on *technical implementability* of the ideas involved. The sorts of questions we will answer are of the form: *Can Yenta be implemented? Do Yentas cluster? Does it help users? Does it appear sufficiently secure?*

A fascinating next step, after answering such questions, would be to investigate how users actually *use* Yenta. For example, the reputation system is likely to generate a set of social conventions, and it is not clear what those will be, and how those will change over time. Such sociological study is also outside our scope here, although the author does hope to do such investigation in the future.

Note that, in investigating Yenta's performance, we make no strong claims here about *optimality.* One could ask a variety of questions about Yenta's clustering methods, either those used to cluster documents within a single Yenta, or the way in which Yentas form clusters on the Internet. What's inappropriate about asking questions about optimality?

- There is no known metric for determining what optimal *means* when deciding whether or not two humans share an interest in a subject, nor in what it means for them to have similar-looking documents. One can invent a large number of definitions, but it's not clear whether this is a useful exercise.
- Optimal solutions usually take a long time to converge; most such problems are NP-complete. Real-world systems always change faster than can be accommodated by such slow methods. In such systems, it is always better to be *acceptably fast*—and approximately correct—than *unreasonably slow*—and perfect.

This lack of concern about optimality is one of many reasons why we make no claims that Savant, which turns documents into keyword vectors, advances the state of the art in information retrieval. Nor do we claim that the algorithm Yenta uses to cluster the resulting vectors—before it contacts other Yentas—into user interests is necessarily an advance, either. In both cases, they are simply *sufficient* to make Yenta useful.

In evaluating results from fielded Yentas, there are a large number of questions we could ask. We shall restrict ourselves to a small set here, but also demonstrate how a large number of different questions could be answered with the infrastructure that is available.

It is important also to keep in mind what we are investigating. We are looking at the *use* of a *particular implementation* of a *single application* that is the exemplar of a *general architecture*. This can answer certain questions, like whether the architecture works at all for *any* application, but also does not answer many others, such as how Yenta might be used differently if it had a slightly different mix of features, or ran on non-UNIX platforms, and so forth.

5.2 Simulation results

We first turn out attention to some simulation results for the clustering algorithm that Yentas use among themselves. This algorithm was described in depth in Section 2.8, with some additional details about the implications of this algorithm in Section 2.9.

Many of these results were also reported in [61] and [58]. (In addition, even earlier results of document clustering—e.g., within a single Yenta—using an older version of the clustering algorithm (not described here) and SMART [188] as the comparison engine, appeared in [56].)

Up to 1000 Yentas were simulated

The Yenta clustering algorithm was simulated for various numbers of interests, typical sizes of its rumor cache, and up to 1000 Yentas, and showed good performance and convergence. Graphical results of these simulations are presented in Figure 17, which have been excerpted from several animations produced to study Yenta's clustering behavior. We discuss the results below.

Three different simulations

Three different simulations are presented. For each, the format of presentation is identical. Each simulation is shown as a series of images taken at various timesteps. The final state of any given simulation is the large image on the right; the six smaller images to the left of that image represent earlier stages of the simulation, reading from left to right and top to bottom.

Interpreting the displays

Each Yenta in any given simulation was given a random interest from the total number of interests available, and then the size of its cluster cache was examined at each simulation step, which indicates how successful it has been at finding other Yentas which share its interest. For all Yentas that share the same pair of parameter values—for example, rumor cache size versus number of Yentas for the first simulation—and are hence in the same bar of the display, the size of their cluster caches were averaged. This average is then compared to the total number of Yentas that *could* have conceivably been in the cluster cache (if all Yentas sharing the interest had been found), and that ratio is expressed as the percentage height of the bar.

Varying the rumor cache

The first simulation shows the effect of varying the size of the rumor cache for up to 1000 Yentas, given 30 different interests split amongst the Yentas. Roughly speaking, it shows that the size of the rumor cache does not make much difference in the speed of cluster formation for more than around 400 Yentas.

Varying the number of interests

The second simulation varies the number of possible interests shared amongst the Yentas with the total number of Yentas, given a rumor cache size of 50. As might be expected, it takes longer to find all the other Yentas one would want as the number of interests increases, or as the number of total Yentas increases.

Ratio of rumor cache to interests

Finally, the third simulation shows the effect of varying the size of the rumor cache for various numbers of interests, given 1000 Yentas. This seems to show that a rumor cache size of 15 is enough for small numbers of interests—between 10 and 30—and that raising this size beyond 35, even for large numbers of interests, does not buy us much.

The basic clustering works

These are strong results. They show that Yenta's clustering behavior is *stable*—Yentas do not try forever to find each other, nor do clusters of them break apart for no good reason—and at least acceptably *efficient*—the number of messages exchanged in order to cluster most of the Yentas is not unreasonable.

5.3 Collecting data from Yenta

Let us now turn to actual application. The general architecture described in Section 2.13 shows how to return data from running Yentas in such a way that it may be analyzed. In summary, the approach is to:

- Run a central server, at an address known to all Yentas, which can collect the data.
- Have each Yenta transmit certain statistical data to the central server, making sure to:
  - Blind the data before transmission by stripping out identifying information
  - Include a per-Yenta unique random number in the data so successive log entries

from the same Yenta may be correlated
    - Encrypt all data during transmission
- Write all the data to disk for later analysis

Collecting this sort of statistical information has considerable risks. If done incorrectly, it could jeopardize users' privacy, and their trust in the entire system, as well as enabling as a single point of attack for a malicious intruder. Therefore, let us follow the steps above, starting with transmission and ending with reception of the data, in order to demonstrate that the collection of this data is not a serious threat.

Central receiver

Each running Yenta knows the address of the central statistics server, and has a task which periodically collects certain information (see Section 5.4 below), creates a logging record, and sends that record to the server. A user's Yenta also sends this information immediately if its user instructs it to shut down.

Blinded data

The information sent is carefully blinded. For example, Yenta by default creates an attestation identifying the user's Yenta by its Yenta-ID, which users may get signed like any other attestation. This attestation is carefully removed from the logging data, before transmission, since otherwise its presence in the data would identify exactly which Yenta logged this record. For more details on the sort of data being logged and why it should be safe, see Section 5.4.

In addition, the statistics-ID which identifies the Yenta doing the logging is a 64-bit random number, having no connection to any other identifier in Yenta. It is communicated only to the logging server—not to other Yentas—and has no personal information embedded in it. Once the data has been written to disk, there is no record of which IP address logged this record, and hence no backpointer to identify where this data came from. All we can know is whether the same Yenta later updates it.

Encrypted transmission

Data being sent to the logging server is encrypted using a session key, in a very similar manner to the way in which Yenta saves its state to disk (see Section 4.8.2). This session key is randomly generated before each attempt to log, and is never reused. A preamble is sent before the actual record consisting of this session key, encrypted with the public key of the logging receiver, which is known by all Yentas. Since only the server knows the private key, only the server may decrypt the session key and thus decrypt the data.

Each individual Yenta keeps track of whether the logging receiver claimed that the logging record was successfully received. If the receiver appears to be down, Yenta simply abandons the attempt to log, remembers that is has done so, and tries again later. This keeps the receiver from potentially being a central bottleneck, whose failure could inhibit the normal operation of all Yentas everywhere. In addition, Yentas periodically prune their logging information if it is too old—this means that, if the receiver vanishes permanently, each Yenta does not store a monotonically-increasing amount of pending logging information.

Vulnerabilities

At the moment, the server decrypts the received data before writing it to disk. It would be slightly safer to leave the data encrypted instead—this would mean that even the server need not know the private key, which could be kept offline and used only when the data is being decrypted for analysis. This complicates analysis, but is being considered. Since all data logged is theoretically already safe—unlikely to compromise users' privacy—the marginal utility of including this step is dubious.

If the server's private key is revealed—by a cracker, say—it will only be useful in attacking a user's privacy if the person who knows the key already has access to the traffic from the Yenta which is the target. After the traffic has arrived at the server is too late—while an attacker who could read the disk could read arbitrary logging

records written there, he or she would have no way to know which machine that Yenta was running on.

Don't do this if you can avoid it

There is no question that running this central statistics receiver is a potentially large invasion of user privacy, and that it presents an inviting target for attack. Any non-research implementation of this architecture should *not* be running such a server. Further, users are well-advised to carefully vet the operation of any architecture which employs such a server—one of many good reasons for ensuring that the source code for any such application is open to public inspection. While the design of the logger was carefully constrained to use only data that appeared safe, it still presents risks.

5.4 What data is collected?

Yenta keeps track of two main classes of things for the benefit of the statistics receiver:

Events

- *Events*. These are changes of state, generally caused by some external action—such as a request by the user for a web page from the user interface, or an incoming connection by some other Yenta. Some events are internally generated, such as a timer expiring indicating that Yenta should rescan the user's documents.

Summaries

- *Summaries* of certain internal state. These are generated on-the-fly, when Yenta has determined that it is time to write a log entry, and are typically estimates of the size of internal data structures.

Counters

In general, any given event will increment a *counter* which keeps track of the number of times which this event has occurred. Some events—such an impending user-commanded shutdown—also cause logging to happen immediately.

The data is persistent

All counters and event logs are maintained in Yenta's permanent state, and are regularly checkpointed to disk. This means that any event which fails to be logged before Yenta is shut down will be logged the next time Yenta is restarted; similarly, counters such as the total number of minutes this Yenta has been in operation will accumulate across successive runs.

A single logging record

An actual logging record thus consists of the following information:

- The statistics-ID (see Section 5.3).
- The time of the message, in Universal Coordinated Time (UTC).
- The current values of all counters.
- The values of all user-settable parameters. These include the various thresholds and preferences the user has set through the interface.
- All attestation strings, and the number of signatures on each attestation. Note that the attestations are stripped of the Yenta-ID normally attached to them, and the signatures themselves are not sent, only a count of them.
- The number of interests known for this user. This is computed from the data structure that keeps track of the Yenta's interests. Note that this is a simple count, *not* the interests themselves.
- The number of clusters this Yenta is in, and the approximate number of known Yentas in each computed in part from cluster cache information. This is not necessarily the total number of interests that this Yenta knows about, since not every interest may have had a cluster found for it yet.
- The number of currently-open network connections to other Yentas. Since a connection is only open when two Yentas have something to say to each other—not simply because they know of each other's presence—this is more an indication of the instantaneous load being placed on the network by this Yenta than it is of how many clusters it is in. (How many network connections have been opened in the past, how many referrals have been done, and so forth, are found in the various counter values mentioned above.)

- A list of events which have transpired since the last log transmission.

Watching one Yenta over time

To actually determine how a particular Yenta has changed over time—such as how rapidly it manages to find clusters for its user's interests—successive records from the same statistics-ID are compared, along with the timestamp logged with each.

A sampling of the sort of counter data which may be collected includes:

- System operations counters: number of startups, shutdowns, errors, bug reports, and time in minutes that this Yenta has been running.
- User interface counters: number of pages and documentation pages fetched.
- Inter-Yenta communication and network statistics: connections initiated and served, protocol opcodes sent and received, network errors, and authentication failures.
- Document clustering counters: number of and total size of documents read, and the number of rescans and reclusters performed.
- Matchmaking, clustering, and messaging counters: number of Yentas encountered, number of clusters joined and left, number of introductions initiated and responded to, the number and total size of individual and cluster messages sent and received.
- Attestation system: number of attestations made and the number fetched, and the number of signatures made and received.

Events that are logged include the following:

- Startup and shutdown
- Contact with another Yenta
- Exchange of cluster information
- Cluster entered or left
- Referral made
- Introduction initiated, granted, or refused
- Message sent or received between users
- Attestation created, signed, or fetched

Clearly, this is a great deal of potential data. We shall examine only a very small subset of it below.

5.5 A sample of results

To evaluate Yenta's performance in the field, a pilot study was undertaken in which Yenta was advertised to a small group of MIT users. This pilot study was deliberately restricted to a relatively small audience, and Yenta's availability was not advertised to a wider audience. The primary reason concerns the implementation of Savant currently present in Yenta—this version of Savant does not have logic to recognize and reject many common artifacts in electronic mail messages, such as included header fields, PGP signature blocks, URL's, and so forth. Thus, it tends to falsely cluster messages based on these machine-generated elements, as well as on their actual content as understood by users. This means that, in addition to clusters that most users would deem useful, there were a large number of clusters which were unhelpful.

A new version of Savant that does not have these disadvantages was made available shortly before this analysis, but not soon enough to facilitate its incorporation into Yenta. Doing such an integration will also change Yenta's understanding of clusters and is not a backwards-compatible change; hence, users in the field will be inconvenienced by having their existing clusters disrupted unless great care is taken. Operational concerns such as this have therefore encouraged only a small deployment at

present so as to minimize disruption of an existing user base. Once the new Savant is integrated, Yenta will leave pilot status and become available to a much wider audience, and will be advertised to such audiences.

5.5.1 Qualitative results

We now turn out attention to some of the available results. This data is derived from the pilot study, in which no more than 50 Yentas were operational at any given time. Exactly how many Yentas are in operation at any instant is somewhat uncertain, for several reasons. We can only know how many Yentas have run recently by investigating the statistics logs, which are keyed by the unique SID. Thus, we must determine whether any given Yenta is still in operation by waiting to see if it continues to log data. In most environments, this would be easy to see, but the particular environment to which Yenta was deployed in the pilot—MIT's Project Athena—tends to encourage users to shut down Yenta frequently, since users rarely have a workstation of their own available and must instead use public ones, which kill background tasks when the user logs out. Finally, the existing Savant implementation in Yenta tends to accumulate too much machine-generated data from email messages. Users tended to discard entire databases and start over on different collections of files when trying to determine which files would best reflect their interests. Since Yenta was not designed to discard its entire database in this fashion, its users took deleting Yenta's saved state and initializing brand-new copies of Yenta, hence artificially inflating the generated statistics. In the analysis that follows, Yentas that do not appear to have run recently have been omitted as having been started briefly and discarded in favor of a new run as a brand-new Yenta. This will be less of a problem with the newer Savant; also, providing users with easier ways to tune Yenta's initial selection of files will help.

After pruning the data for various artifacts such as these, and to reduce the analysis task somewhat, we were left with a sample size of 21 Yentas. This sample will be used in the discussion that follows.

Clustering works

In general, results from fielded Yentas bear out the simulation results in Section 5.2. For example, Yentas will cluster correctly if they share sufficiently-close interests, and, likewise, they will correctly conclude that they should *not* cluster if their interests are divergent. This is the case despite the technique, as described in Chapter 2.8.3, of mixing in other data from the local Yenta's rumor cache to provide plausible deniability for its user to a querying Yenta.

Yentas can find each other

Yentas have demonstrated that they can find each other in all the ways designed into the architecture—via the bootstrap server, via broadcast on the local Ethernet segment, and by detecting the presence of a formerly-unknown Yenta from the contents of some other Yenta's rumor- or cluster caches.

The protocol works

Further, the running Yentas do not display serious protocol abnormalities—any given pair of Yentas that were formerly unknown to each other initiates a conversation, dumps interests back and forth, and correctly clusters, or not, based on those. They do not get hung up exchanging data forever, and correctly revisit each other at various intervals to see if anything has changed.

Message and attestation relaying works

Yentas which share an interest can correctly relay messages back and forth to each other. This behavior was verified both in one-to-one messaging and in one-to-cluster situations. Similarly, attestations may be created, signed, and displayed to Yenta's users.

Determining the user's initial interests has an obvious path to improvement

Yenta's determination of user interests was judged subjectively by investigating the interests that it found from a variety of files. In the currently-fielded Yenta, its determination was sufficient, but not as good as it can be. In large part, this is due to its use of an older version of the Savant comparison engine, as detailed above.

Saving state works

Individual Yentas can correctly save and restore their state across crashes—either of Yenta or the underlying machine—and across user-commanded shutdowns. They have also been shown to interoperate with a variety of common web browsers.

Yenta is fast enough

Yenta has also proven to be acceptably fast. Even running on five-year-old hardware (an HP 9000/725), it can scan and cluster several megabytes of mail—about as much as is reasonable to use—in a handful of minutes. A typical clustering attempt with another Yenta, in which both Yentas must share not only their interests but many interests from their rumor caches for plausible deniability (see Section 2.8.3), takes a few minutes. In part, this is due to the throttling effect of the network, but it is also the case that we do not wish Yenta to consume all available CPU resources on the machine on which it runs—after all, it runs as a background task most of the time.

Since Yenta is designed to run with only occasional user attention, even these results are better than they appear. For example, even though it takes a few minutes for two Yentas to determine whether or not they share an interest, the user can still fetch pages from the user interface, talk to other Yentas already known to be a shared cluster, and so forth.

Logging errors helps a lot

Handling internal errors and reporting them to the statistics server was very useful in the field. The very first deployment of Yenta to users turned up a number of minor bugs, generally caused because users tended to use Yenta slightly differently than its implementors sometimes did—that caused tasks to occasionally err. The symptom of such a failure is generally that the user sees a page request of the user interface simply hang until it is retried; this starts a new task, and generally whatever bug was encountered would not be retriggered. However, because such failures were reported to the statistics server, complete with backtraces, tracking down the bugs and fixing them was *much* simpler than it would have been had self-reports from the field been the only method.

5.5.2 Quantitative results

To lend some concreteness to the discussion above, let us examine just a few selected statistics from those logged by Yentas in the field. These statistics cover 21 Yentas deemed representative, logged over a period of about 25 days, from the pilot study. They are drawn from approximately 2200 individual entries to the statistics logger.

The table in Figure 16 below summarizes the results. We investigated a few elements from several different areas of Yenta's operation: how the user interface was used; how many documents were scanned and how many interests were determined as a result; how the attestation system was used; some clustering data; a quick look at Yenta's networking protocol, and how long Yentas tended to run. For each such element, we present the total across the *n=21* Yentas, the minimum and maximum values seen, and their average and standard deviation. Many of the minimum values are zero, generally due to a Yenta being started, minimally configured, and then shut down without rerunning it later. Approximately 3 Yentas from the sample below show a short enough total runtime that this is likely for them, but their results were included in the totals because there was still useful data—such as number of documents scanned and number of interests found—from such Yentas, even though they were not allowed to continue running long enough to do anything useful for their users.

The statistics above show that users made extensive use of the UI—in other words, they interacted a lot with their Yentas. They also fetched a large number of help pages, which is to be expected of a new application. One user scanned a very large number of documents (over 8000), although must scanned a must more reasonable number (the median was around 400, and the average around 600). From these, users were typically presented dozens to a hundred or so interests, and tended to find at least a few other clusters to join. A typical Yenta sent a bit more than a megabyte—spread

Parameter	Sum	Min	Max	Average	Std dev
UI pages fetched	2925	11	648	139.2	143.1
Help pages fetched	264	1	32	12.5	6.9
Documents scanned	12773	0	8388	608.2	1779.5
Number of interests	2592	0	1406	123.4	312.9
Signatures verified	353	0	75	16.8	22.5
Clusters joined	89	0	50	4.2	10.9
IY opcodes sent	3032	0	811	144.3	216.2
IY kilobytes sent	28854	0	10770	1374.3	2715.9
Minutes of operation	117719	0	34122	5605.7	10287.1

Figure 16: Some selected statistics from fielded Yentas.

out over the three weeks of the pilot—to accomplish this level of clustering. Finally, any given Yenta typically accumulated around 100 hours of operation in this interval—being generally shut down when its user was logged out, for those using MIT Project Athena machines—although some ran almost the entire time and are still running as of this writing.

5.6 Security

Let us now turn to evaluating Yenta's security. It is widely accepted that there is no way to be absolutely sure that any particular piece of software, if at all complicated, is completely secure. However, there are many potential ways to increase our confidence, which include, among others:

- *Black-box analysis.* This involves attempting to crack Yenta's security completely from the outside, as if it was a black box.
- *Formal methods.* These involve proving theorems about the underlying cryptographic operators, *and* about how they are used in Yenta's actual implementation.
- *Design review.* This involves examining the overall principles of the architecture advanced in Chapter 2 and Chapter 3, and combining that with the description in Chapter 4 of the actual application fielded.
- *Code review.* This involves actually reading the code and looking for weaknesses.

While it is certainly *possible* that someone will subject Yenta to black-box analysis, we have no intention of doing so here; there seem to be much better options at our disposal. And, unfortunately, formal methods are quite attractive, but typically are not feasible for entire applications. They can be quite helpful in evaluating particular network protocols (such as SSL) or particular cryptographic functions (such as DES), but are less likely to reveal whether a particular application correctly implements the design which has been formally analyzed, due to the time and effort required to do rigorous analysis of a large body of code. They can also miss incorrect design assumptions, such as incompleteness of the threat model.

Yenta's design, and the design of the architecture of which it is a part, are public information. This encourages review. In addition, the actual source code of Yenta is also available, for a number of reasons, including pedagogy, increasing the portability of the application, and the presumption that openly-available code is itself a social good. However, one of the most compelling reasons to make code for an application such as Yenta public is to increase the chances that others will find weaknesses.

The strategy chosen for Yenta is twofold:

- Make it easy to vet Yenta's code
- Give people incentives to do so

The first of these is partially accomplished by Yvette, as described in Section 3.4.4. Briefly, Yvette encourages collaboration among people who are interested in evaluating a large body of code, by enabling them to divide up the work, write reviews of small sections, and review the work of others. Yvette also enables those who are less skilled to nonetheless peruse the reviews, by showing how much of the entire corpus of code has been reviewed and, for sections that have received several reviews, whether those reviews have been generally positive or negative.

There are several possible incentives for others to review Yenta's code. Making Yenta more secure is clearly a social good, at least among those reviewers who share the author's political agenda. Further, as is commonly the case in software projects whose source is publicly available, those who make particularly important contributions either to the code or its review are often rewarded by improvements to their reputation in that social group.

Yenta also tries directly to appeal to other programmers for review. The following rather long insert is an excerpt from the web pages which announce Yenta, and is indicative of the sort of things we are asking others to look for:

> **Please help improve Yenta's security, so that all of its users may benefit.** We are offering incentives for finding major flaws. To be most helpful to us, and hence to do the most to improve Yenta's security, please read *all* of the topics below. They cover:
>
> - How to comment on the code.
> - What's in it for you.
> - What counts as a flaw.
>
> **Commenting on the source code**
>
> Your easiest starting point is probably to *critique Yenta's source code directly.* Yenta's current source code is available via Yvette, which allows *collaborative critique* of a body of code: each person may make comments on a single function, a whole file, or an entire subtree of the source, and others may view these comments. This allows dividing up the work.
>
> Since it is expected that most possible flaws will concern some well-defined area of the source code, you should remark on it at the appropriate point in the source tree that Yvette gives you. *If you think you have found something particularly serious*, you may want to send mail to bug-yenta@media.mit.edu telling us what you found. Please see also our description of what counts as a flaw.
>
> **What incentives we have for you**
>
> There are several incentives available to encourage people to improve Yenta's security:
>
> • **Community good.** This is worth doing for its own sake, because you are helping everyone who uses Yenta to be able to use a system that will not inadvertently expose personal information, will not crash, and will be useful to its users.
>
> • **Public recognition.** All comments about Yenta that have been given to Yvette are available to everyone to read. Particularly insightful comments may also be mentioned in various acknowledgments when papers about Yenta are published.
>
> • **Goodies.** If you are the *first* to report a particularly serious security problem in Yenta, we'll give you something. If you're local, this might be dinner. If you're not

local, it might be something else appropriate. If you care strongly about getting something, then please *comment in Yvette* (if there is a particular area of the code that is affected), *and* remember to send us mail. Please note that our judgment of what counts as "serious" is absolutely at our discretion. But don't worry—we'll be fair. It is quite probable that there are things missing from the description below (perhaps we forgot one of the cases that don't count in the threat model description); this doesn't mean we owe you dinner if you find something we don't think it a major flaw, but which we didn't mention. On the other hand, we'd still like to hear about it—if nothing else, to correct our description.

What counts as a flaw?

This is a description of our **threat model.** In other words, what sorts of flaws are we looking for?

Security bugs versus other bugs

• **We're interested in *all* bugs...** so please, if you spot something in the source which is a bug in *functionality*, even if it does not have security implications, please comment about it in the source and also send us mail at bug-yenta@media.mit.edu. If you trip over a bug while using Yenta, but don't know where it is in the source, send us mail and at least let us know.

• **...but we're most interested here in *security* bugs.** Not only are undetected security bugs dangerous to users, but they are likely to go unreported unless someone actively looks for them. After all, a bug in functionality, such as Yenta crashing, or doing the wrong thing with a command, is likely to be noticed by the user who experiences it, but a security bug could be totally silent and yet deprive all users of their privacy.

What sort of attacks are we talking about?

• *Things which don't count.*

  • **Denial of service doesn't count.** In other words, if someone can arrange to make your Yenta do *nothing*, either by overloading it, running it out of resources, or attacking the connection of its machine to the net, that's outside of the scope of what Yenta is designed to survive. Of course, if you see a simple way to prevent a denial of service which is *specific to Yenta*, please let us know.

  • **Careless users don't count.** Users who deliberately choose poor passphrases will compromise their own security. Yenta can't stop them. Similarly, even though Yenta takes care to arrange a very strong SSL connection between the user's browser and Yenta itself, if the user is running their web browser with an insecure connection *between their keyboard and their web browser*, Yenta cannot possibly know this, and cannot prevent it from occurring. This can easily happen if the user is using X—with the keyboard and screen on one machine, and Yenta running another—and is not using SSH or some similar protocol between the two machines. Similarly, if the user is running a crippled browser that supports only 40-bit session keys, Yenta *is* willing to talk to the browser, but this connection is only secure against attackers without many resources.

  • **Attacks by root on the same machine don't count.** A superuser on someone's workstation can read any bit of memory, can substitute compromised versions of binaries for formerly good ones, can install trojan horses that capture every keystroke the user types before it gets to any application, and so forth. Yenta cannot hope to avoid such attacks. Note in particular that Yenta is *more* vulnerable to a memory-sniffing attack than programs like PGP, because Yenta must remember the user's private key at all times—PGP need only remember it for the instant that the session key is being encrypted. And any attack that compromises the binary—whether on the local workstation, or by altering NFS data if the binary is fetched over the network—also cannot be countered.

  • **Byzantine failures don't count.** In other words, if you surrounded some in-

nocent victim's machines with *only* machines running bogus, compromised versions of Yenta that are all under your control, you could certainly figure out what the user is interested in, and probably do a lot worse damage as well. Yenta explicitly assumes that all the rest of the Yentas on the net are *not* evil. One or two is okay, but a vast majority is not.

• **Savant index files don't count.** Yenta stores its crunched, vectorized information about your mail in a binary but unencrypted form on disk, in your ~/.Yenta directory. Although the directory is read-protected against all but its owner, this is not secure against an attacker who can read the filesystem. Since this information originally came from plaintext files which are *also* in the filesystem, it is assumed that this approach does not compromise the user's privacy any more than it already was. Note that Yenta's other saved state, such as the user's private key, his stored conversations, and so forth, *are* encrypted and never appear on disk in the clear, even for a moment.

• **Attacks on the maintainers' machines don't count.** Even though the distributions are cryptographically signed, and even though the source code is available via Yvette, there is certainly the potential for corrupting the actual code being distributed, by attacking the machines upon which Yenta is developed. While it would be possible to secure these machines better, doing so gets in the way of getting work done, and Yenta *is* a research project. So you are not allowed to attack the actual source—*or* our machines!—and then claim a victory. Just don't.

• **Traffic analysis doesn't count—yet.** The current version of Yenta uses point-to-point IP connections when passing a message from one Yenta to another. Later releases will employ a broadcast-flood algorithm, either by default or on request, to make it harder to tell where the real endpoints are of a communication. This makes it more difficult for an attacker who cannot monitor every link in real time to know which pairs of Yentas are exchanging a lot of traffic (and hence which may have users who are interested in the same things).

• *Things which do count.*

• **Problems in Yenta's cryptography.** This could be insecure encryption modes, vulnerabilities in the protocols used between Yentas or in the way that permanent state is stored on disk, and so forth.

• **User confusion that leads to exposure.** If Yenta does something that causes a user to be a confused and inadvertently reveal something that he did not wish to, that is a bug. But this must be *Yenta's* doing—not a con game perpetrated by another user, for example.

• **Failures of anonymity.** Yenta tries to keep the connection between an individual user's true identity and his Yenta identity unknown, unless the user deliberately divulges that information. If there are easy ways to defeat this, we need to know. However, see the note about traffic analysis *not* counting, so far—a later release will fix this.

• **Spoofing.** If one user can masquerade as another, complete with valid-looking attestations which are fraudulent, this is certainly a bug.

• **Missing items from the description on this page.** We may be missing examples, either in the listing above of threats which Yenta is just not designed to handle, or in this listing of possible places to look for problems. If so, please let us know, so we can update the list. This helps two sets of people: Yenta's users, who get a more accurate picture of what Yenta can and cannot do, and Yenta's reviewers, who won't waste their time investigating a vulnerability which we consider to be outside of Yenta's scope.

5.7 Risk analysis

Let us now turn our attention to an analysis of Yenta's residual risks, using the criteria in the previous section as guidelines. Where are the weak links? If Yenta is to be trusted, is it actually trustworthy?

5.7.1 Denial of service

Certainly the most obvious weak link in the design is that of *denial of service*. We have explicitly stated that this is *not* a problem we are trying to address, but how well do we sidestep it anyway? Let us first ignore any denial of service which takes down the actual host upon which an individual Yenta is running on, or its network connections anywhere else. We shall also assume at the moment that we are talking about an attack on a *single user's* Yenta—not on all Yentas. Assuming that the underlying host and its network are functional, how vulnerable is Yenta to a denial-of-service attack?

Single Yentas

There are several ways to mount such an attack. One involves simply opening a connection to a Yenta and giving it an infinite list of possible interests, or making one particular interest of infinite length. Yenta throttles its reception of any network connection to a fixed maximum number of bytes per task timeslice, hence other tasks will not be starved even if another Yenta attempts to monopolize its attention. In addition, Yenta will start dropping interests if the list from any given Yenta is too large, and will drop additional vectors of any one interest if it exceeds a threshold.

It is certainly possible to make Yenta's rumor- or cluster-caches useless by inventing a very large number of unique-seeming Yentas—e.g., for example, with a single process that keeps claiming to have a different Yenta-ID for every connection—and then bombarding a stream of connections. This can certainly fill the rumor cache with a large amount of junk, making this particular Yenta's referrals useless to *other* Yentas. It can also fill the rumor cache with *known* junk, hence compromising the digital mix described in Section 2.8.3; we shall have more to say about that below.

If the attacker can deduce the local Yenta's interests accurately enough, such a bombardment might conceivably also fill its cluster cache with entries which all correspond to the attacker's identities. This can effectively cut off the Yenta from *real* clusters that share its interests, and resembles the case of a Byzantine attack, but mounted from a single host. Whether or not this attack can succeed also depends on whether the local Yenta is using the attestation system to reject other Yentas which do not have appropriately-signed attestations.

Defending against such attacks can be quite difficult. One easy solution, which is not currently implemented in Yenta but which would be quite simple to do, involves throttling the number of unique Yentas accepted from any given IP address in some time interval—for example, no more than 100 unique YID's from any given IP address in a month. This raises the workfactor for the attacker considerably, since the attacker must now control many more hosts. [IP spoofing is not a reasonable approach for the attacker, since the communications protocol depends upon two-way TCP traffic, including a cryptographic challenge, which means the attacker must see return packets. And yet we have also assumed that the host's underlying network is working; this means that routing is working and the attacker therefore cannot simply be sitting on the host's local interface, or on the local wire, and modifying all packets—this counts as the network *not* working.] Note that we cannot throttle the number of YID's per IP address to only *one* unless we wish to break the ability to use Yenta on a timesharing system—if we were to do this, every Yenta connecting to the local Yenta from the timesharing host, except the first would be dropped as an attack.

All Yentas

Denying service to *all* Yentas is a trickier task. Assuming that the distribution is not corruptible—meaning that both the signatures on the distribution and the evaluations in Yvette are secure—one possibility would be to spuriously advertise some critical problem in Yenta to its user community, perhaps via a mailing list. If this ever becomes a problem, all mail from the maintainer to Yenta's users will have to be digitally signed so they may check it for authenticity. At the moment, this is a fair amount of overhead, so such messages remain unsigned.

Because both the debugging logger and the statistics receiver are expected to occasionally be down, all Yentas can cope with the results and will not fail no matter how long these servers are down. Hence, even a total failure of these servers will not bring down all Yentas. Further, since the communication from Yentas to the central server is essentially one-way at the level of the Scheme protocol sent, there is little opportunity for the server to freeze a Yenta via an inappropriate response.

It *is* possible that there is some way to subvert the SSL implementation of the statistics server—e.g., at a protocol level below the actual Scheme forms exchanged—such that it causes a Yenta which is trying to log statistical information to freeze—for example, by exploiting some bug in the implementation which causes SSL negotiation to hang if the server simply halts at the appropriate moment. If the C code of the SSL implementation is frozen, Yenta will freeze, because no other tasks will ever run. Since all Yentas eventually attempt to log to this server, this will freeze them all. It is currently unknown whether such a vulnerability in the SSL implementation exists. A possible solution, if it *does* exist, would be to wrap a timeout around all SSL session negotiations and simply abort any tasks whose timeout expires. This can cause each Yentas to become momentarily sluggish each time it tries to log, but this should not be a major performance problem if the timeout is not excessive.

5.7.2 Integrity and confidentiality—protocols

As discussed in Section 5.6 above, it is believed that the most serious risk to either integrity or confidentiality of the data exchanged by Yentas is that of poor security practices by its users. This ranges from running Yenta on compromised machines to picking poor passphrases to using web browsers which only allow 40-bit keys to typing passphrases or otherwise using the user interface across insecure links—e.g., running a web browser via X and then using an insecure connection between the X server and the X client. Such mistakes and poor practices are incredibly common and very difficult to guard against—for example, it is essentially impossible to know for sure, from a program's point of via, whether any given X connection is or is not secure, since the program must know more about the environment and the threat model than can be sensibly expected. Similarly, while Yenta *can* trivially simply refuse to talk to any web browser which fails to use encryption with enough bits in its keys, one can make the argument that this might needlessly disenfranchise users who are using Yenta in a way that even 40-bit browser keys are perfectly acceptable—such as the case wherein the browser is running on the same host as the Yenta and no bits are traversing the network.

The problem of weak passphrases

Yenta does not currently make any attempt to ensure that the passphrase chosen by the user is at all secure. This would be a relatively simple addition, but raises important concerns about users forgetting passphrases if they are forced to be long. Most users find themselves unable to remember an 80 to 160-bit string, even expressed as a passphrase of random words, on first sight. (It is commonly accepted that most humans can only commit about one bit of information per second to long-term memory; this has obvious implications for the long-term memorability of a newly-generated passphrase which is random enough to be unguessable by someone else.)

Maybe users should write them down?

The best solution to the passphrase issue might actually be to *encourage users to write down their passphrases* somewhere, such as on a scrap of paper in a location known to the user. This is heresy to the traditional security establishment, but certain threat models may make it sensible. For example, many users may be in an environment where local users can become root on their workstation (e.g., system administrators) and no passphrase will protect such users. However, nonlocal users may be arbitrarily distant and may have no idea where the user is physically. Given such an attacker, a written-down passphrase is no less secure than one which is not—but writ-

ing down the passphrase may encourage the user to pick one that is sufficiently long as to have a useful number of random bits in it.

5.7.3 Integrity and confidentiality—spies

Assuming that the basic cryptographic protocols are adequate, and that the user is using Yenta safely—no insecure browser connections, a good passphrase, and so forth—there are still underlying issues of confidentiality in particular. Consider the denial-of-service attack described above in Section 5.7.1, in which a single Yenta is targeted by an effectively unlimited number of bogus other Yentas, all under control of a single attacker. Whether or not Yenta throttles unique YID's per IP-address and unit time, there is some combination of resources which is guaranteed to cause an arbitrary proportion of the local Yenta's communications to *all* be with the attacker's Yentas. At this point, the local Yenta has been captured and is in a case explicitly disallowed by our criteria in Section 3.2.4. How bad is the damage?

In the simplest case, this attack breaks the digital mix described in Section 2.8.3. This means that, when the local Yenta exchanges interests with the attacker's Yentas, any interest which does not come from the attacker's supplied interests is known to belong to the local Yenta. This means that these interests are no longer plausibly deniable.

Trusted attestations may be the only feasible solution

This is quite a difficult attack to defend against. We cannot even attempt to spread information by insisting that third-party Yentas do the comparison of each interest, and then collating their responses, because all such third-party Yentas are themselves still under control of the attacker. It appears that the only obvious solution to this problem is to have the local Yenta insist that *every* Yenta which it talks to possess some attestation signed by a party which can be reliably known to *not* be the attacker. Of course, this is very likely to dramatically reduce the number of other Yentas that will be listened to by the local Yenta, perhaps to zero, but there seems little choice—if *everyone* you talk to is lying to you, and yet you feel compelled to tell *someone* of your interest in something, you are in trouble. Your only alternative may to figure out how to get someone you already trust to vouch for someone else.

5.7.4 Contagion

Is a network of Yentas vulnerable to contagion? Such an outcome could allow a malicious attacker to disable the entire system; it also allows cases in which the system might simply fail all on its own, by accident.

While it does not *appear* that there is any potential for such a thing, bizarre failures of this type have been seen in the past in other systems [121]. Yenta never accepts any code fragments from elsewhere, which should minimize the chances of a true virus being able to propagate. For example, when reading a Scheme form from a network connection, Yenta uses a custom-written parser that disallows almost all Scheme forms except lists, strings, numbers, and booleans. This guards against an attack which is otherwise possible against both Common Lisp and Yenta's particular version of Scheme, in which the attacker uses the #. reader macro—which means evaluate this form at *read* time, not load or compile time—to cause the machine parsing the form to execute arbitrary code. [For example, if Common Lisp calls *read* on the form *(+ 2 3 #.(malicious-code-here) 5)*, it will execute malicious-code-here before reading the rest of the form. Even if *eval* is not called on the result of the form (and hence the addition is not performed), the malicious code will have been already run. Common Lisp has the *with-standard-io-environment* form, which will inhibit #., but SCM does not and hence requires a home-grown solution to this problem.]

Yenta does not use this custom-written, safe parser when reading forms from the file in which is saves its permanent state. However, since this file is encrypted, an attacker would have to break the encryption to cause Yenta to execute arbitrary code, which seems like a much more difficult problem than simply causing the user to run the wrong application via a wide variety of easy attacks involving subverted hosts and the

like. Hence, attempting to propagate a virus in this manner would require manually compromising each host in order for it to succeed, at which point it can hardly be said to be a virus at all.

Deliberate shutdown

There was some thought given, early in the Yenta project, to having a global shutdown code installed in at least early versions of Yenta as released. Such a code would be intended to halt *all* Yentas reliably, in case the Yenta network protocol behaved badly and threatened the usability of the network infrastructure. The intended method of action would be to have each Yenta first broadcast the shutdown code to all neighbors, and to then halt for, e.g., no less than a week, before allowing itself to be run again. The code itself would be cryptographically signed using a private key known only to the implementors, and whose public half would be installed in every Yenta. It would contain an expiration date, after which any Yenta would ignore the stop code, such that it could not permanently kill all Yentas forever. And, to be extra safe, the code would presumably be implemented using a threshold scheme, such that several individuals would have to collaborate to reconstruct the required key to emit the code. Not only would this guard against an unfortunate mistake, but having several of the individuals be in different sovereign countries would aid in preventing a duress attack, in which the implementor was forced through legal or extra-legal means to disable the Yenta network—presumably by some government actor that wished to suppress anonymous encrypted speech.

Such a shutdown scheme would be a deliberately-installed method of destroying the Yenta network, at least for a time, due to an intentional contagion. Early results from Yenta indicated that the potential for a network meltdown due to Yenta was low, while the hazard of ever having such a mechanism installed in Yenta was high. Given this, and the implementation work required to install such a feature—and to verify that it *would* act correctly when triggered but would *not* trigger falsely—the feature was intentionally omitted from the fielded system.

It is nonetheless still entirely possible that there exists some pathological interest, message, or attestation which will be propagated to other Yentas and which causes any Yenta possessing it to malfunction. Such an outcome is exactly analogous to the ARPAnet collapse described in RFC528. No such mechanism is currently known. It is hoped that careful code review, for example via Yvette, may discover any such mechanisms *before* they are accidentally triggered. It is also hoped that such a malfunction will at least allow logging data to be returned to the implementor; this might allow the issuance of an updated version before all Yentas are crippled. However, in most scenarios it may be that the logged data would be insufficient—since interests, messages, and many attestations are *not* returned to the logging receiver, a pattern-dependent pathology in them will not be returned. Only when the implementor's Yenta failed would the actual pathological case be made available for inspection.

5.7.5 Central servers

Every central server in the fielded Yentas represents a vulnerability. As discussed in Figure 2.13, for example, the mere existence of the statistics receiver represents a great risk of accidental information disclosure if Yenta logs some identifiable information by mistake. In addition, the existence of such a server represents a user-perception risk—some users may not be interested in any protestations that such a design is safe, may distrust it on principle, and may not user Yenta for that reason. Given the sorry results from trusting similar sorts of assertions in other systems, it would be hard to blame them for such a stance.

The bootstrap server also represents a small risk. In particular, a malicious takeover of the server could cause all newly-started Yentas to be forced to talk to a particular set of other Yentas—presumably those under the control of whoever took over the bootstrap server. This is not guaranteed to work, since each Yenta that starts first broad-

casts on the local wire and only ask the central server if not enough responses are received, but it may succeed against Yentas that start in environments where few other Yentas are already running. Similarly, someone who can control answers on the local wire can subvert a newly-started Yenta into only using a particular set; since such an attack can only affect new Yentas on a particular wire segment, it has less potential for widespread mischief than taking over the bootstrap server.

The debugging server presents few risks, save that it is possible that a bug in Yenta's implementation—such as logging the value of some variable that could reveal something about the user's interests or the contents of conversations—may leak private information. However, Yenta's use of this server is rare. Because communication with this server is strictly one-way, from Yentas in the field to the server, it seems unlikely that taking over the server could accomplish much besides inconveniencing the implementors (and, secondarily, making it impossible for brand-new Yenta users to automatically sign up for the couple of mailing lists which talk about Yenta; they could still do so manually even in this case).

5.7.6 Nontechnical risks

Yenta faces some nontechnical risks which might also impact its utility. For example, how exactly to use the attestation system—what might be useful to say about oneself, for instance—has been left deliberately underspecified. In part, this is an experiment to see how people *do* decide to use the attestation system, but it may backfire—without sufficient guidance, users may not use it at all, or they may use it in such idiosyncratic ways that attempting to use the attestation system for *automatic* filtering of incoming messages becomes very difficult. (This is especially true given the rather user-unfriendly choice in current Yentas of requiring such filtering to user regular expressions; regexps are *not* expected to be understood by most users and it is hoped that a later version of Yenta will use something friendlier. How exactly to do this is a matter of some research.) Note that, even if users cannot use regexps in any useful way to automatically reject particular Yentas—hence leading perhaps to a spam problem—they may still manually add Yentas to their rejection lists, thus killing spam from any Yenta that has sent it even once. They may also, of course, still read attestations themselves and use their own judgment about whether to accept an introduction or a message from someone based on their own reading.

It is conceivable that Yenta may run afoul of patent issues. This is generally a problem in software systems these days, and especially problematic with those employing cryptography. It is also rumored to be a method of attack from corporate interests against free software generally, given that most authors of free software do not have legal counsel and certainly to do not have the war chest of patents that large companies tend to have. This is, alas, a risk that is not unique to Yenta.

Because Yenta facilitates anonymous, private speech, it is likely to irritate many governmental and even nongovernmental actors who have vested interests is discouraging such speech. (For example, the European Union has recently proposed—though not yet adopted—prohibitions against electronic anonymous speech [48]. This issue comes up frequently in the United States as well, despite its Constitutional protection in other media [65][120].) However, barring changes in existing law, especially in the United States where Yenta is being developed and fielded, it seems unlikely that excessive coercive force could be applied either to Yenta's users or to its implementors.

5.8 Other applications of this architecture

Let us now turn our attention to a brief evaluation of how the underlying architecture employed by Yenta might be used in several other applications. The questions we are answering here are: How well does the architecture support these applications? Where might the architecture need to be extended? Compared to more traditional ways of

implementing these applications, does this architecture offer unique advantages? This discussion is necessarily speculative—none of these sample applications have been implemented, although many of them would not be difficult to do.

Web pages

One example would be an application might use the contents of web pages that have been fetched by the user as the input to the document comparison engine, rather than the contents of the user's own files or mail as is currently done in Yenta. This application bears some resemblance to the Webhound/Webdoggie system [103], although it is actually a superset—not only is it distributed, unlike Webhound, but Yenta incorporates an interpersonal communication system which Webhound lacks.

Building such an application seems relatively easy. The minimal-work approach would be to use some other external program, such as the *wget* program, to fetch all web pages in the user's bookmark list, and then simply point Yenta at the resulting collection of files. Another approach, more convenient for the user but slightly more work for the implementor, would add the necessary code to Yenta to enable it to fetch web pages directly and feed them into Savant. Whichever approach is chosen, performance would probably be improved if the Savant comparison engine was augmented to understand more about the structure of the web page—such as attempting to compare web pages by number of outbound links to foreign sites, or number of included images, and so forth—because the Webhound/Webdoggie research showed that doing so improved the performance of that system as well. (Clearly, simply importing the relevant part of Webhound's page-comparison code would be a straightforward way to go about this.)

This application seems well-adapted to the architecture described in Chapter 2. It has considerably advantages over the original Webhound implementation, because users no longer have to worry about some central site knowing which pages they browse, and also enables them to easily share information about web pages by simply talking to each other—Webhound only suggested pages, with no explanation and no easy way of contacting the other user(s) who may have seen those pages already.

Database queries

Another example is an application that attempts to build groups of people who do similar database searches—a sort of community-builder that might be used within a single company that does database mining. Such an application could help inform those working in this hypothetical company about other groups or divisions which seem to be duplicating work, or which allow people doing similar searches to pool their resources. Implementing this application requires removing Yenta's existing document comparison engine—Savant—and implementing some new comparator which, given two database queries, can compute some similarity metric between them. It would also require some trivial modifications to change the printed representation that Yenta uses to describe an interest from a short vector of keywords to, perhaps, the actual database query that was issued.

Assuming that it is possible to develop some metric that can suitably compare database searches, then this application, too, should work well given the framework of Chapter 2. If used only within a company, the anonymity and privacy features presented could well be overkill, but perhaps not—intracompany politics and competition can sometimes be ugly. And the attestation system might be used to ensure that no information is somehow shared with rivals—consider a system in which the agents only talk to others which display attestations that have been properly signed by some well-known entity in the company, such as its human-resources department. This turns the web-of-trust architecture presented by the attestation system into the substrate for a more hierarchical, certificate-authority-based system, and enables a high degree of trust that any given agent really *does* belong to a user who works for the company. Properly done, this assurance can be much strong that trusting to a firewall

or to the domain-name system, both of which tend to be easy to subvert in practice [128][161].

Romantic matchmaking

What changes would be involved in making Yenta a true *romantic* matchmaker—an application that was explicitly designed for dating? On the surface, this seems both obvious, simple, and well-adapted to Yenta—for example, the attestation system might serve very well in helping to controllably share certain crucial information between prospects, while the underlying cryptographic security and nym system can help to control undesired information flows before partners commit to a physical meeting, if they ever do.

But a closer look shows that this is not quite the problem that the original Yenta was designed to solve, for a number of reasons. Yenta assumes that shared interests are sufficient to bring people together, but romantic matchmakers cannot make that assumption—indeed the phrase opposites attract may be quite relevant for many users. In part because of this, romantic matchmakers often require each user to specify a profile which describes an intended match, and this profile may bear little resemblance to the user creating it. This lack of self-similarity—we may be attempting to match users based on characteristics they do not share—breaks a naive implementation of the clustering algorithm described in Section 2.8. In addition, a handmade profile may lack the ability to do hillclimbing, because such profiles often consist of very few words (e.g., 10 or 20) and not the large number of words—and hence long vectors—that document summaries such as Savant tend to generate. One possible solution to this might be to instruct potential users to instead pick, say, online works of one sort of another—web pages, book chapters, and so forth—that could be of interest to a potential mate. A profile-creation step which requests large amounts of information in ways that a comparison engine can partially order may also help; this would require careful thought and correctly-structured data. But how do we deal with the opposites-attract problem?

For concreteness, let us name our potential users *Harry* and *Sally*, and consider some ways out of this dilemma if Harry is looking for a mate who is *unlike* him, and Sally is also. Assume that Harry wants someone who is outgoing and friendly, but is himself curmudgeonly; likewise, assume that Sally *is* outgoing and friendly, but wants a curmudgeon. They would be perfectly matched, if only they could find each other. However, if Harry creates a profile that looks for outgoing, friendly people, and Sally finds it, she will incorrectly assume that Harry *himself* is not a curmudgeon, and will reject the proposed match.

One way out of this might be to modify Yenta such that it understands explicitly the connection between *pairs* of interest clusters, such that Harry Yenta can cluster itself into a clump of other curmudgeons, while simultaneously clustering itself into an outgoing-and-friendly cluster. Sally's Yenta could presumably do the same. If both Yentas then understood the *meaning* of each finding themselves in both clusters simultaneously, and if each Yenta kept track of which cluster represented a profile of its *own* user and which cluster represented a profile of the mate being sought, then it is possible that the opposites-attract problem might be solved. It does not seem, at least at first glance, that this is a prohibitively difficult programming project, although it does seem to be the sort of thing that might require extensive tuning and a careful user interface so as not to confuse its users.

Electronic commerce

Let us now consider implementing some sort of ecommerce system, in which buyers and sellers wish to find each other in order to exchange goods. The first order of business involves creating some sort of comparison metric that can translate some description of goods or services into something upon which a partial order of similarity may be imposed; otherwise, clustering cannot use hillclimbing. One potential

approach to this problem, depending on the domain, might be to embed products in a hierarchy of related products, and to measure similarity between two types of products by comparing the distance between points in the hierarchy. This might allow clustering to build groups of buyers and sellers which are interested in similar products.

The next issue concerns what to do once a cluster has been formed. One approach might be to have buyers and sellers simply broadcast messages, using the messaging system described in Section 2.10. The message sent may either be human-to-human, as in Yenta, or algorithmically generated, for example some sort of open-outcry bidding system—there are no doubt a large number of potential algorithms which could take advantage of such an architecture. Once a buyer/seller pair are aware of each other's existence, they may of course also simply send messages directly to each other, again as human-to-human or in some sort of automated bidding algorithm.

The attestation system could be used to excellent effect in such a system. For example, buyers who are happy with the seller's performance may volunteer to sign attestations from the seller which verify that the seller is trustworthy. (Recall that attestations, being kept by their owner, will presumably not be kept if uncomplimentary, and hence buyers will be unlikely to be able to say negative things about sellers because sellers will not offer such attestations in the first place; see Section 2.11.) Buyers might themselves have attestations which sellers can sign—perhaps as part of some other element of the transaction—indicating that the buyer has paid for prior purchases.

The support of this architecture for private, authenticated message exchange, combined with the attestation system, makes the architecture described in Chapter 2 an attractive choice as the substrate for an ecommerce system. The most difficult part of the design which is unique to the architecture—as opposed to, say, which bidding strategies to use and so forth—is likely to be support for forming clusters in the first place. If there is no natural landscape of similarity which can be used to support the hillclimbing, this approach may not be acceptably efficient.

5.9 Motivating adoption of the technology

Designing, implementing, and fielding a decentralized application has many challenges. While doing so can have important benefits for users, it can be significantly more difficult for implementors than a centralized system, for a number of reasons:

Challenges

- It is much more difficult to update a large number of applications in the field than a single, central application. This implies that the system must be closer to production quality—not alpha or beta—before first ship.
- Because the application must be significantly more robust at first release than is often observed, it may take longer for a given development staff to field such a system than many centralized systems. For uses, such as in businesses, where time to market is the dominant factor, such a delay may be a major liability.
- The application may have to be more complex to correctly handle the inevitable mix of versions that will be present in the field.
- There exists a significant issue of *education*, of the user base and of others who must talk about or interact with the system, because truly decentralized systems are still unusual. During deployment of Yenta, for example, even sophisticated users continued to ask, "Where's the server?" repeatedly until they understand how the system operates.

Solutions

These issues need not be fatal. For example, many centralized systems such as web-based ecommerce sites rely on an already-implemented toolkit, for example the Apache web server [7], and do a relatively small amount of additional work to add whatever functionality is required to make their sites into a business. A similar,

widely-available toolkit for building decentralized systems—such as a commercial-strength infrastructure that implements the basic architecture described in Chapter 2 and Chapter 3—could go a long way towards enabling rapid implementation and deployment of such decentralized systems. In addition, such a prebuilt toolkit might help to avoid some of the more common errors in the implementation and use of cryptographic algorithms and protocols, since the work required to design and verify them can be spread across multiple applications and multiple reviewers. While it is still possible for some other part of the deployed system to compromise the otherwise-good crypto, at least there is that much less of the design and implementation that must be written and checked.

Motivating businesses

A more serious concern, at least for business users, is the large value of data-mining to many businesses and their reliance upon such data as a revenue stream. Indeed, with the falling price of computers, some businesses are willing to give away a computer valued at several hundred dollars *for free* in return for the ability to collect vast amounts of detailed personal information from their users as the computer is being used [85]. In this case, even if a decentralized system offers technical advantages, such as robustness and insulation from the labor of answering subpoenas, and even if the system is viewed favorably by users, the business must forgo a revenue source in order to be socially responsible. This is a tradeoff that few businesses are apparently willing to undertake.

Clearly, if the financial motivation is sufficiently large, almost all businesses will ignore any scheme that protects their users' privacy. Such a motivation would have to factor in the possible loss of goodwill from customers, the time and effort required to answer subpoenas, and the possibility of enforcement action from government actors.

Thus, the solution for motivating businesses to do the right thing—in this case, protecting the privacy of their users—must eventually come down to making it too expensive for them to violate their privacy instead. This is very unlikely to be a purely technical solution, given the example above where it is obvious that detailed information about particular users may be quite valuable commercially. Instead, businesses must either lose customers, and hence revenue stream, to others which are more protective of their customers' civil rights, or they must be forced to be more protective via legal action.

Given a scenario in which a privacy-protecting system gets to market at approximately the same time, and costs a business less—for the various robustness reasons mentioned elsewhere, for example—it is also in the interest of that business to *educate its customer base* about why they are getting a superior deal in terms of their civil rights. Such an education and advertising campaign, if properly handled, may pay off by discouraging customers from using competing systems that are not so protective. While it is historically true that such campaigns are difficult and often do not motivate a large proportion of users, it is always possible that such attitudes will change—for example, if well-publicized privacy disasters continue to emerge.

Legal remedies are another option. At the moment, the United States in particular is in a poor position in protecting privacy rights, as discussed in Chapter 1. One reason is certainly the lack of public awareness of the problem. Another may be the sense that the situation is in some way inevitable—that the use of computers to handle personal information *must necessarily* lead to reduced privacy. It is hoped that the results of this research will serve as an example that this need not always be so. If this example becomes widely known, it may influence legislative attitudes by making it obvious that many businesses have no technological justification for their actions. This may thus lead to legal pressure on businesses and other actors for the protection of their users' rights.

5.10 Future work

Having a large number of Yentas in operation provides several intriguing opportunities for further study. We shall investigate some of these here.

5.10.1 Sociological study

Any new technology can benefit from studying the way in which people use it. Yenta, in particular, is an unusual combination of matchmaking service, mail system, collection of newsgroups, document summarizer, and reputation system, among others.

One obvious approach involves exploring the sorts of groups that form, and whether users find that they deliberately include or exclude certain types of documents to try to find particular such groups. Since there is no toplevel ontology of which groups exist, the prevailing social structure is more like the one that exists in everyday, non-networked life—one cannot simply ask, "What are all the interest groups in the world that I might possibly become a member of?" because there is no such central registry. Yenta shares the same characteristics. Yet users who hear through channels outside of Yenta about particular groups may be tempted to try to join them. If Yenta does not support this explicitly, users will likely find a way—but how?

Yet another possibly-fruitful direction concerns the reputation system. What will people say about themselves? What will (and won't) get signed by others? What social signalling systems will evolve? Will these systems span clusters or not? What sorts of filters will people write to take advantage of the reputation system—or will they use it only to evaluate potential conversational partners? What are the patterns of signatures—can we infer anything about social organization by who signs whose attestations? The range of possible questions is very large, but could be sociologically interesting to answer.

5.10.2 Political evaluation

Yenta also has a political dimension. Will it change the way people tend to think of privacy and computer-based processing of personal information? Will it influence systems designers to take civil liberties more into account? Will the decentralized nature of the architecture lead to more such architectures, even in cases where it is, for example, robustness, and not privacy, that is most at issue? Will the transparency goals for vetting its source code—particular Yvette—lead to other projects being easier to evaluate collaboratively?

All of these questions are good ones, and it is hoped that they will be the subject of future research.

5.11 Summary

In this chapter, we have evaluated the architecture via simulation, and demonstrated that it scales to realistic sizes and performs well. We then described how to instrument the sample application so it could be analyzed, and discussed qualitative and quantitative data from a pilot deployment, which show that the application as fielded performs acceptably, and provides guidance on how to improve it. We then investigated some residual risks of the architecture and the application, including some exploration of how to defend against attacks that we declared to be outside of our original threat model. We have speculated on the methods that might be required to motivate business users to adopt the technology, despite current financial incentives to the contrary. Finally, we briefly mentioned some intriguing directions for future work.

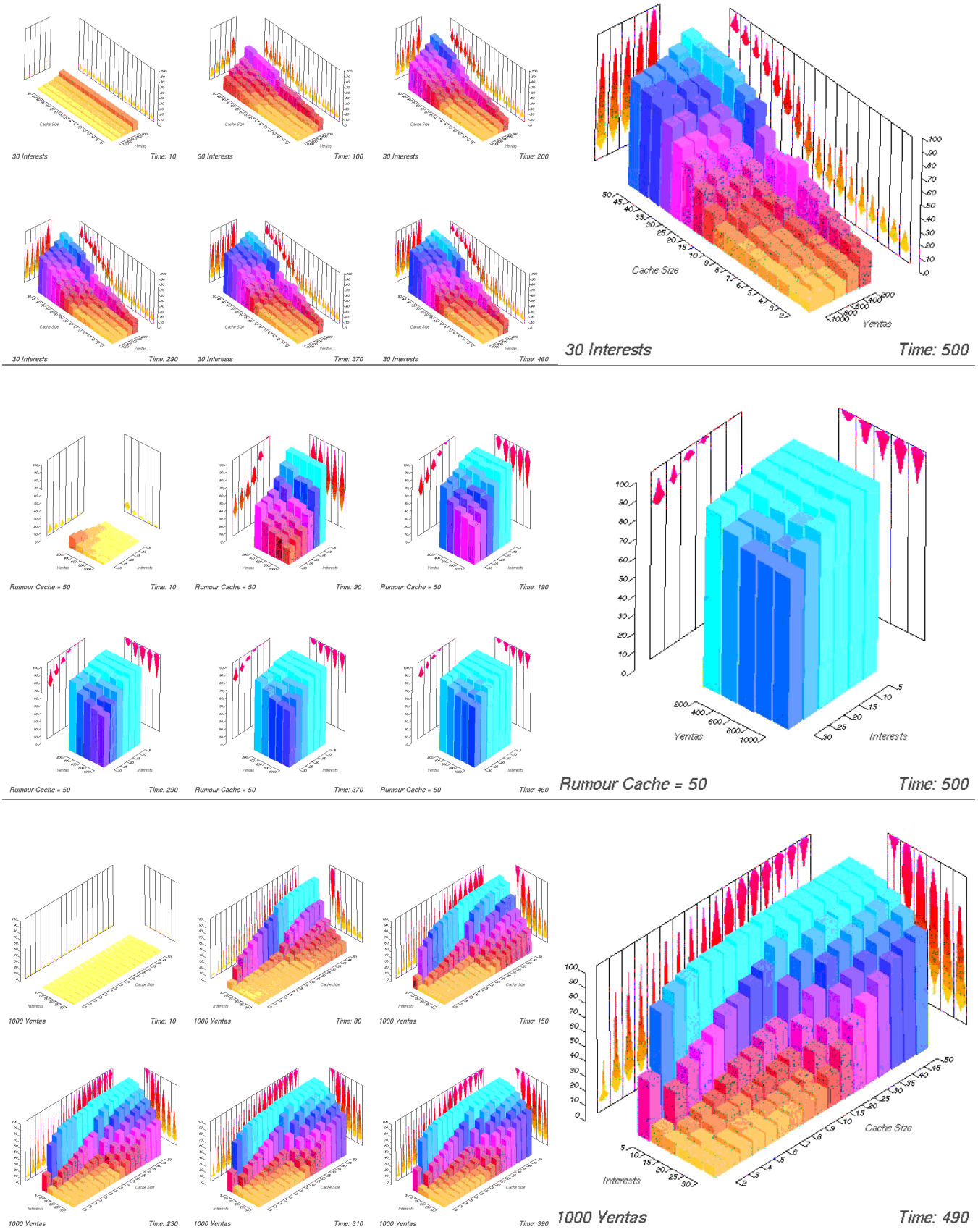

Figure 17: Simulation results. See text for details.

CHAPTER 6 *Related Work*

6.1 Introduction

We have presented a general architecture and a sample application, and some evaluation of them, designed to promote a particular sociopolitical agenda and to demonstrate that starting with such an agenda can lead to technological advances. Let us now turn our attention to related work in some relevant fields.

This research touches on a large number of possible topics. We shall restrict ourselves here to examining:

- Other types of matchmaking systems — Section 6.2
- Other decentralized systems — Section 6.3
- Other systems and software that have been designed for political purposes — Section 6.4

6.2 Matchmakers

In general, systems that perform any sort of matchmaking task are *centralized* systems. Such an organization has several useful advantages, especially to the implementors of such systems, if they do not also have to deal with personal information:

Why centralization is a popular approach

- They are easier to administer—all, or almost all, of the relevant software can run on hosts directly under the administrative control of the implementors
- If they are being used for a business, it is often obvious how to structure the system such that users may be charged fees, or have advertising delivered to them as they use the system
- If the business model of the matchmaker *also* requires that personal information be reused for other purposes—such as marketing—then centralizing all data on the company's own servers makes this easy.

Collaborative filtering

Webhound/Webdoggie [103] and HOMR/Ringo/Firefly [112], for example, are typical examples of centralized matchmakers. A central server maintains information about user interests, and users connect to the server (in both cases, with web browsers) to discover whether they have a match. Both systems require the user to be proactive in establishing and maintaining an interest profile, although Webhound/Webdoggie also obtained leverage by using a data source the user already kept updated, namely his or her hotlist.

Brokering services

Kuokka and Harada [99][100] describe a system that matches advertisements and requests from users and hence serves as a brokering service. Also a centralized server, their system assumes a highly-structured representation of user interests.

Sixdegrees

Sixdegrees [110] is an interesting idea in matchmaking, generally for professional reasons; this site keeps track of *who you already know* and uses this information to find minimal spanning trees to others who you would like to know. It does this by asking for email addresses corresponding to others that you know, and also for their relationships to you (as well as other information, such as their profession), and then contacts those people to see if they agree. If they do not repudiate the relationship, the system records the correspondence. Users are always identified; unlike most other systems, there are no pseudonyms. Users can then ask queries such as, "Who do I know who knows a lawyer?"

The system is somewhat cumbersome because of the need to involve everyone explicitly (anyone you name must take the effort to become a member themselves), but its narrow targeting of social relationships makes it likely to find interesting contacts. It is, of course, another centralized system, although it takes certain efforts both to reassure its users that their information will remain private—although, of course, they make no assurances about either crackers or subpoenas— and that the system cannot easily be *gamed* to expose large numbers of relationships—for example, you can only find out about the relationships of other people to people you already know, out to a very limited diameter, and can only spam those you already know, which is presumably not very productive.

PlanetAll

PlanetAll [150] takes a somewhat different approach. It concentrates on *finding people you once knew*, rather than on finding new people you might like to know. Like Sixdegrees, it is a centralized, web-based service, and everyone using the service is identified by their real name. Unlike Sixdegrees, the primary organizing principle behind PlanetAll is *affinity groups.* Such groups are prespecified, named entities corresponding to organizations in the real world—not online—of which the user was at one time a member. They are typically schools, clubs, or religious organizations, and PlanetAll allows one to search for them by keyword. When registering, the user specifies affinity groups, and is then notified when others join the group. He or she can send messages into the group or to particular individuals.

Spamming is prohibited by the rules of service, and, since individuals are always strongly identified, tracking them down and barring them is easy. On the other hand, it is not clear what would happen if someone who was never part of some affinity group in real life were to join one anyway—such a party crasher would probably simply be tolerated, as least if he or she was not obnoxious, because everyone else in the group might assume that *someone* knew them.

One can also tell PlanetAll about particular individuals in the system and ask it to send mail when that individual's information (such as work address) changes. It is presumed that individuals already know each other when they receive notification of one joining the group—thus, PlanetAll concentrates on finding people after one has lost track of them, rather than on describing unknowns to each other. PlanetAll also has a number of other interesting features. For example, it allows users to enter their travel itineraries, and will notify them when their paths cross in foreign cities.

As with Sixdegrees, PlanetAll users must trust that the central site will protect their personal information. Since such information could be valuable to a number of commercial interests, and also to those contemplating identity theft, this could be a major exposure.

Romantic matchmakers

Although romantic matchmaking is *not* an explicit goal of Yenta, there are a large number of matchmaking systems specialized for this application, and they are worth studying. Such systems appear to be invariably centralized. For example, Match.Com [41] is a straightforward romantic-matchmaking service. Users fill out a form detailing their own characteristics and those of people they would like to meet (sex, age, geographic location, etc.), which are used in a simple match/filter algorithm; they also post personal ads to supply more detail once a user's filter has selected some ads. Similarly, the Jewish Matchmaker [43] (unfortunately also called Yenta, for obvious reasons) is one of several more-specialized systems that function similarly: surveys for filtering, personals for secondary selection, and a centralized server, all backed up by a web-based interface.

A rare decentralized example

Kautz, Milewski, and Selman [91] are one group, of very few, to have taken a more distributed approach to matchmaking. They report work on a prototype system for expertise location in a large company. Their prototype assumes that users can identify who else might be a suitable contact, and use agents to automate the referral-chaining process. They include simulated results showing how the length and accuracy of the resulting referral chains are affected by the number of simulated users and the accuracy and helpfulness of their recommendations. Yenta differs from this approach in using ubiquitous user data to infer interests, rather than explicitly asking about expertise. In addition, Yenta assumes that the individuals involved probably don't already know each other, and may have interests that they wish to keep private from at least some subset of other users.

6.3 Decentralized systems

There are a variety of other decentralized systems that bear consideration here. For the most part, these systems may be divided by their underlying metaphors: *biological*, *market-based*, or *other*. We shall discuss all three below.

Both biological and market-based systems are often used in the allocation of scarce resources, although with a difference in emphasis. For example, biological systems often model individual actors or agents through their births, lives, and deaths. It is commonly assumed that the characteristics of agents change relatively slowly over their lifetimes, but that an entire population may change through evolution. Individual agents generally have very limited models of the world and sometimes vanishingly small reasoning abilities. Market-based systems, on the other hand, tend to assume agents which exist for indefinite spans of time, but can change their behavior relatively quickly due to learning within an agent. In addition, information flows—as opposed to flows of matter—are often considered to dominate the interaction, and explicit negotiation between agents with high levels of reasoning are common.

Biological metaphors

The *artificial life* approach is explicitly informed by a biological metaphor [94]. This discipline tends to model systems as small collections of local state that have generally been mapped into a simulation of some physical space. Within this space, these bundles of state may interact solely through local interactions—there is no action at a distance. Systems modeled often tend also to simulate real biological systems, albeit simplified versions—ant and termite colonies [142], predator/prey systems and various simulations of Darwinian or Lamarckian evolution [19][102], learning [57][62], immune systems [95], and many more. Some simulate decidedly nonbiological systems using biological metaphors—for example, many problems in optimization are often effective solved using genetic algorithms [96]; for example, producing optimal sorting networks [79].

The choice of self-contained bundles of state, and strictly local communication, stems naturally from systems which either simulate or are inspired by the natural world, where nonlocal effects tend to be rare. Most such systems run on uniprocessors, but there are exceptions. For example, many learning [62] or simulated-evolution [165]

systems have been implemented on SIMD or MIMD architectures such as the CM-2 or CM-5 Connection Machines from Thinking Machines. Others have been distributed to collections of uniprocessors connected via the Internet. One example is *NetTierra* [139], a network-based implementation of the original *Tierra* [138], a system originally written to explore the evolution of RNA-based life via an easy-to-mutate machine language.

Market-based metaphors

Market-based approaches tends to use negotiation, barter, and intermediate representations of value—such as money—to enable a collection of actors to decide on individual strategies [25][111]. One example of such a system is *Harvest* [74], which uses a decentralized collection of *gatherer*, *broker*, *collector*, and *cache* elements to greatly improve the performance of, e.g., web servers. Element use market-based ideas to decide how to allocate various resources such as storage or bandwidth.

Consider also a system in which we have a *heap*, such as that found in a Lisp system, where objects point at each other. Reclaiming unused space in a heap is called *garbage collection*, and doing so if the heap spans multiple machines can be quite slow due to communications overhead. Using a market-based approach, in which storage essentially *pays rent* and storage which runs out of money is deallocated [40] can make this problem much more tractable by keeping almost all the computation required local to individual machines.

Other approaches

Not all decentralized systems necessarily require either competition or cooperation between agents—some simply use decentralization to achieve pure parallelism, turning a network of uniprocessor CPU's into an emulation of a MIMD multiprocessor. One common example of this these days is *cryptographic key cracking* [32], in which thousands of CPU's participate in searching the keyspace of a particular encrypted communication. This application is typically *political* in nature—in general, participants take part in order to help demonstrate that ciphers such as 56-bit DES are woefully insecure [15][24][34][42][76][184].

6.4 Political software and systems

Let us now examine various software systems that have been designed with a particular eye towards their political environment. We will concentrate here only on systems which attempt to advance what we believe to be the *socially responsible* position in our political argument—and not, for example, systems such as the centralized Intelligent Transportation Systems described in Section 1.4.

Pretty Good Privacy

By far the most famous example of such software is *Pretty Good Privacy*, or *PGP* [187]. PGP is one of the most widely-used strong-cryptography packages in the world. Recent versions have even been deliberately exported from the United States, even though doing so electronically is illegal. Instead, the First Amendment to the US Constitution was exploited as a loophole—it has already been determined that printed books are not subject to regulation under US export-control law. Thus, source code was printed into a ten-volume book, which *is* legal to export, in a format that was explicitly designed to be easy to scan and convert back into electronic form overseas. (Since then, other important cryptographic efforts have been exported in the same way—for example, all of the VHDL and loader code describing how to build a hardware DES-cracking machine was printed in machine-scannable form expressly to allow this [42].)

PGP's development was motivated by explicitly political aims—its author, Philip Zimmerman, wrote it to make strong cryptography easily available to the masses, or at least to those masses who owned personal computers. And since then, it has become a lightning rod for discussion concerning US cryptographic-export policy.

PGP itself does not depend on any sort of network infrastructure—it encrypts and decrypts files only. However, it is most useful when combined with a network, rather

than when being used to mail encrypted floppies back and forth. Various popular mail-handling programs, such as Eudora for Macs and PC's, and Mailcrypt for GNU Emacs, have incorporated it into their design.

Anonymous remailers

Other political software has made the network a more explicit part of their design. Consider anonymous remailers [10][23][66], which are designed to hide the origin and destination of messages being sent from one computer to another. They work by encrypting messages in transit, and routing them through a large number of computers in various political domains. The assumption is that no single entity could successfully compromise every computer and every network link in the chain, and that this lack of total surveillance will allow truly-anonymous information exchange.

The contents of such messages are varied. Many concern topics which are potentially embarrassing or dangerous to those discussing them, such as unusual lifestyles, or discussion of medical problems such as HIV which might cause the discussant to lose his or her job or social standing. Others are explicitly political in nature, sent by people living in regions where political dissent can lead to imprisonment or execution [11].

anon.penet.fi

One particularly famous remailer was the *anon.penet.fi* remailer [77], run by Johan Helsingius. This service offered single-hop anonymity—messages sent to this remailer had identifying information stripped out, but were then delivered as usual to their destination. This made it particularly easy to use without the special software often required of multihop *Mixmaster* [10][23][66] remailers. It also offered *nyms*—one could have a stable, pseudonymous identity through the use of this service, rather than being completely anonymous. Anyone could reply to a message posted through anon.penet.fi, back to the original author, even though both parties would not know each other's actual identities.

This mechanism also led to a certain amount of insecurity. For example, in one well-publicized case in 1995, the Church of Scientology was able to get the local government in Finland to subpoena the site's operator for the mapping between one particular nym and the real email address of the person behind it. In 1996, the Church tried to determine if a particular individual had ever used the service. The site was eventually shut down by its operator, who cited the increasing load on his time that running it required, and the availability of at least partial substitutes elsewhere on the net.

The Anonymizer

Consider now the *Anonymizer*, which attempts to make it possible to fetch web pages without informing the web server of the identity of the machine doing the fetching—presumably for use in reading pages with controversial content, or to deny marketers the ability to target the reader for profiling. It is a single, centralized server, and simply proxies requests through itself, rewriting HTML links such that following a link on a fetched page will go back through the Anonymizer. While it can effectively hide users from sites, it is useless against traffic analysis attacks—it operates at a single, well-known address and from a single point of presence. This makes its communications easy to tap, either at the site or by looking for requests from a given user to the Anonymizer itself. Even if SSL is in use, thus hiding the actual URL's being requested and the contents of the pages returned, traffic analysis at the user's site can instantly reveal that the Anonymizer is in use at all, and even this is often sufficient to target the user for various unfortunate consequences. Further, sites which offer content may deliberately deny content to the Anonymizer, to force users to come from well-identified IP addresses. Finally, users of the Anonymizer must trust that the site really *is* honoring its stated policies of not keeping logs of the traffic through itself.

Crowds

A more-sophisticated system, developed after the Anonymizer, is the *Crowds* system [141]. This system is also an attempt to strip identifying information from web surfers, and uses decentralization to foil traffic analysis. Participating users join a

crowd—a collection of other machines, all of which participate in the system, and which randomly reforward HTTP requests and responses among themselves before sending them to their final destinations. This means that any particular web page fetched by a user could come from any of the participating machines at random, hence denying the web server the ability to know precisely who is fetching which pages.

This system is explicitly aware of the problems of traffic analysis, both at the web server itself and in the intervening links between that server and the user, and takes steps to foil it. It also reduces the problems of trusting the privacy policy of a single site.

Web filters

Web-filtering programs grew directly out of political concerns—they are software packages which are deliberately designed to block content from particular users, generally minors and anyone else who might be coerced into using them, such as library patrons in some cases. Some of them, such as RSACi [140], rely on self-ratings by sites. Others, such as PICS [177], rely on third-party ratings. These third-party ratings may be either public, and possibly distributed, or provided by the manufacturer of the filtering software, and often private.

Since *someone* must choose which sites are acceptable and which are not, there is an implicit political agenda to using such software. Even systems which claim to allow the user to select any other third party's recommendations may be abused given enough control of the network infrastructure. For example, China carefully controls traffic across its borders, and could insist that all web surfers use only government-approved PICS sites for their filter lists. In addition, those systems in which the vendor of the filtering software choose are often extremely heavy-handed about what sorts of sites are deemed unacceptable. In response to this, Bennett Haselton [75] has spent considerable time and effort exposing the antics of filter manufacturers who claim to be blocking "sexual content" but are also blocking a wide variety of nonsexual web sites that happen to have politics that the filter vendors find unacceptable. The list of sites blocked by these packages are secret, ostensibly for reasons of competitive commercial advantage, but this means that there is virtually no oversight for what often turns into an appalling censorial exercise.

FSF and Open Source

Finally, let us consider an *intellectual property methodology,* as opposed to particular systems or programs. The methodology of interest is the union of the *Free Software Foundation* and the more-recent *Open Source* movement. Both of these approaches view *freely-redistributable software* as a social good. While they differ on the details of what this means and how to achieve it, they are in substantial agreement that the *freedom to examine and modify source code* is the cornerstone of building high-quality software. Many famous examples of their effort exist, such as the GNU collection of hundreds of utility programs—Emacs, autoconf, automake, gtar, gmake, and all the rest—and other projects which use their licensing terms but were not written by the FSF—such as Linux, SCM, and so forth.

Both the FSF and the Open Source group have an explicit political agenda, which they enforce through the technology of copyright and contract law. Thus, their technology is that of intellectual property per se, rather than that of software itself. Their efforts have had an enormous effect on the way that software is currently developed, especially—but not exclusively—that which runs under various varieties of UNIX, and is likely to leave a considerable legacy.

6.5 Summary

In this chapter, we have touched briefly upon matchmakers, decentralized systems, and politics. All three of these fields are assuming increasing importance as the Internet continues to expand and its user base continues to grow. The research that led to

Yenta and its underlying architecture did not arise from the vacuum. Instead, it is explicitly informed from—and, in some cases, in reaction to—some of the existing systems and methods of practice currently popular in the field.

CHAPTER 7 *Conclusions*

In Chapter 1, we presented a rationale for the particular agenda—personal privacy—that drove the architectural design in Chapter 2, the security issues addressed in Chapter 3, and the sample application in Chapter 4. In Chapter 5, we demonstrated that the result appears to meet its design goals without forcing privacy and functionality to be traded off against each other. Finally, in Chapter 6, we investigated some work related to this research, exploring other matchmakers, decentralized systems, and explicitly political systems. In this chapter, we shall draw some general conclusions about the work presented here.

We have demonstrated that starting from a social or political agenda can have wide-ranging effects on how technology is designed. This research has shown that carefully protecting the personal privacy of users in a broad class of applications can lead to a design which is technically superior in several respects—indeed, the decentralized nature of the design, its reliance on strong cryptography, and many other design elements seemed inevitable once the underlying agenda was chosen. The result indicates that, for a large class of potential applications, protecting privacy does not necessarily require that one make a tradeoff between privacy and either robustness or functionality.

It is hoped that the widespread availability of this research will lead to commercial and legislative chnages in several viewpoints, including

- the relation of technology to issues of personal privacy, and
- the utility of strong cryptography.

This is, after all, the fundamental goal of the research—while Yenta is an important and useful application, and while its use can serve as both an advertisement for the underlying concepts and also help solve real problems for real users, it is the architecture and the rationale for its development which are the most important issues here.

As discussed in Section 5.9, one of the enduring issues concerns getting those with financial incentives in their users' lack of privacy to adopt the technology discussed here. The Yenta application itself, by serving as an example and by raising awareness of the issues, may help in this regard. However, it is only the first step along a long but ultimately rewarding path.

References

[1] Mark S. Ackerman and Laysia Palen, "The Zephyr Help Instance: Promoting Ongoing Activity in a CSCW System," *Proceedings of the Association of Computing Machinery's Conference on Human Factors in Computing Systems (CHI '96)*, pp. 269-275, ACM Press, Vancouver, British Columbia, Canada, 1996.

[2] Allan Robert Adler, ed., *Litigation under the Federal Open Government Laws, 19th edition*, American Civil Liberties Union Foundation, 1995.

[3] Philip E. Agre, mod., *RRE@weber.ucsd.edu* mailing list,{http://dlis.gseis.ucla.edu/people/pagre/rre.html}

[4] Philip E.Agre and Marc Rotenberg, eds., *Technology and Privacy: The New Landscape*, MIT Press, 1997.

[5] Ellen Alderman and Caroline Kennedy, *The Right to Privacy*, Knopf, 1995.

[6] R. J. Anderson, "Why Cryptosystems Fail," *Communications of the ACM*, 37(11):32-40, November 1994.

[7] The Apache Group, {http://www.apache.org}

[8] Julian Assange, admin, *Best-of-Security@suburbia.net* mailing list archives.

[9] Associated Press newswire, April 17, 1998.

[10] Andre Bacard, *Anonymous Remailer FAQ*, {http://www.well.com/user/abacard/remail.html}

[11] Patrick Ball, Paul Kobrak, and Herbert F. Spirer, *State Violence in Guatemala, 1960-1996: A Quantitative Reflection*, American Association for the Advancement of Science, Washington, DC, 1999.

[12] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transport Protocol -- HTTP/1.0," RFC1945, May 1996 {ftp://ftp.isi.edu/in-notes/rfc1945.txt}

[13] Marshall Berman, *All That is Solid Melts Into Air*, Simon & Schuster, 1982.

[14] Henry Campbell Black, Joseph R. Nolan, and Martina N. Alibrandi, *Black's Law Dictionary*, West/Wadsworth, 1990.

[15] Matt Blaze, Whitfield Diffie, Ronald L. Rivest, Bruce Schneier, Tsotomu Shimomura, Eric Thompson, and Michael Wiener, "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security," *Report by an Ad-Hoc Group of*

Cryptographers and Computer Scientists, Business Software Alliance meeting, Chicago, IL, November 20, 1995.

[16] Jeffrey Bradshaw, ed., *Software Agents*, AAAI/MIT Press, Spring 1996.

[17] Anne Wells Branscomb, *Who Owns Information? From Privacy to Public Access*, BasicBooks, a division of HarperCollins Publishers, Inc., 1994.

[18] Louis M. Branscomb and James H. Keller, *Converging Infrastructures: Intelligent Transportation and the National Information Infrastructure*, the Harvard Information Infrastructure Project, MIT Press, 1996.

[19] Rodney A. Brooks and Pattie Maes, eds., *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, MIT Press, 1994.

[20] Amy Bruckman, "Gender Swapping on the Internet," *INET '93*, San Francisco, August 1993.

[21] *CALEA: Communications Assistance for Law Enforcement Act*, 47 U.S.C. §§ 1001, et seq. (1994).

[22] Ron Canetti, Cynthia Dwork, Moni Naor, and Rafi Ostrovsky, "Deniable Encryption," *Proceedings of the Workshop on Security in Communications Networks*, Hotel Cappuccini, Amalfi, Italy, September 16-17, 1996 {http://www.unisa.it/SCN96/papers/CDNO.ps}

[23] David Chaum, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Communications of the ACM*, 23(2), February, 1981.

[24] D. D. Clark, *Computers at Risk: Safe Computing in the Information Age*, [Final report of the System Security Study Committee], National Research Council, National Academy Press, 1990.

[25] Scott Clearwater, *Market-Based Control: A Paradigm for Distributed Resource Allocation*, World Scientific Publishing, 1996.

[26] C|NET news.com, April 16, 1999, {http://www.news.com/News/Item/0%2C4%2C0-34889%2C00.html?st.ne.180.head}

[27] Michael Coen, "SodaBot: A Software Agent Environment and Construction System," MIT AI Lab TR 1493, June 1994.

[28] Philip R. Cohen and Hector J. Levesque, "Communicative Actions for Artificial Agents," [a KQML critique], *Proceedings of the First International Conference on Multiagent Systems*, Victor Lesser, MIT Press, 1995.

[29] Gary Cornell and Cay Horstmann, *Core Java*, SunSoft Press, 1996.

[30] Barry Crabtree and Nick Jennings, eds., *Proceedings of the First International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology*, The Practical Application Company, 1996.

[31] Kevin Crowston and Thomas Malone, "Intelligent Software Agents," *BYTE*, pp.267-271, December 1988.

[32] Matt Curtin and Justin Dolske, "A Brute Force Search of DES Keyspace," *;login: The Usenix Magazine*, May 1998.

[33] Pavel Curtis, "Mudding: Social Phenomena in Text-Based Virtual Realities," *Proceedings of DIAC '92*.

[34] K. Dam, *Cryptography's Role in Securing the Information Society (aka the CRISIS report)*, [Final report of the Cryptographic Policy Study Committee], National Research Council, National Academy Press, 1996.

[35] Drew Dean, Edward W. Felten, and Dan S. Wallach, “Java Security: From HotJava to Netscape and Beyond,” *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 6-8, 1996.

[36] C. A. Dellafera, M. W. Eichin, R. S. French, D. C. Jedlinksy, J. T. Kohl, and W. E. Sommerfeld, “The Zephyr Notification System,” *Proceedings of the Winter 1988 Usenix Technical Conference*, February 1988, pp. 213-219.

[37] Y. Demazeau and J. P. Müller, eds., *Decentralized AI, Volumes 1, 2, and 3*, North-Holland, 1991, 1992, and 1993.

[38] Keith Decker, Katia Sycara, and Mike Williamson, “Modeling Information Agents: Advertisements, Organizational Roles, and Dynamic Behavior,” *AAAI-96 Workshop on Agent Modelin*g, March 1996.

[39] L. Dent, J. McDermott Boticario, T. Mitchell, and D. Zabowski, “A Personal Learning Apprentice,” *Proceedings of the National Conference on Artificial Intelligence*, MIT Press, 1992.

[40] K. Eric Drexler and Mark S. Miller, “Incentive Engineering for Computational Resource Management,” *The Ecology of Computation*, B. A. Huberman, ed., Elsevier Science Publishers, B. V. (North-Holland), 1988.

[41] Electric Classifieds, Inc., {http://www.match.com/}.

[42] Electronic Frontier Foundation, *Cracking DES: Secrets of Encryption Research, Wiretap Politics, & Chip Design—How Federal Agencies Subvert Privacy*, O’Reilly & Associates, Inc., May 1998.

[43] Cupid Networks, Inc., {http://www.yenta.email.net/}

[44] Jonathan Emord, *Freedom, Technology, and the First Amendment*, Pacific Research Institute for Public Policy, 1991.

[45] Oren Etzioni and Thomas Mitchell, *Proceedings of the AAAI Spring Symposium on Interface Agents*, Stanford, 1994.

[46] Oren Etzioni and Daniel Weld, “A Softbot-Based Interface to the Internet,” *Communications of the ACM*, July 1994.

[47] European Communities, *Directive on the protection of personal data (9546/EC)*, {http://europa.eu.int/comm/dg15/en/media/dataprot/news/925.htm}, {http://europa.eu.int/comm/dg15/en/media/dataprot/law/dir9546.htm}

[48] EU Report A4-0141/99 on the draft Joint Action, adopted by the Council on the basis of Article K.3 of the Treaty on European Union, to combat child pornography on the Internet (10850/5/98 - C4-0674/98 - 98/0917(CNS)), Committee on Civil Liberties and Internal Affairs, Rapporteur: Gerhard Schmid.

[49] Remy Evard, “Collaborative Networked Communication: MUDs as Systems Tools,” *Proceedings of the Seventh Systems Administration Conference* (LISA VII), USENIX, Monterey, CA, November 1993.

[50] Export Administration Regulations, 15 C. F. R. §§730-74.

[51] Dave Farber, mod., *Interesting-People@eff.org* mailing list.

[52] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” RFC2068, January 1997 {ftp://ftp.isi.edu/in-notes/rfc2068.txt}

[53] Paul Feldman and Silvio Micali, “Optimal Algorithms for Byzantine Agreement,” *20th STOC*, pp.148-161, ACM, New York, 1988.

[54] Leonard N. Foner, “Clustering and Information Sharing in an Ecology of Cooperating Agents,” *AAAI Spring Symposium on Information Gathering from Distributed, Heterogeneous Environments*, Knoblock & Levy, eds., Stanford, CA, March, 1995.

[55] Leonard N. Foner, “Clustering and Information Sharing in an Ecology of Cooperating Agents, or How to Gossip without Spilling the Beans,” *Proceedings of the Conference on Computers, Freedom, and Privacy '95 Student Paper Winner,* Burlingame, CA, 1995.

[56] Leonard N. Foner, “A Multi-Agent System for Matchmaking,” *Proceedings of the First International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology,* Barry Crabtree and Nick Jennings, eds., The Practical Application Company, 1996.

[57] Leonard N. Foner, “Paying Attention to What’s Important: Using Focus of Attention to Improve Unsupervised Learning,” MIT Media lab SM Thesis, June 1994.

[58] Leonard N. Foner, “A Security Architecture for a Multi-Agent Matchmaker,” *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS ‘96),* Keihanna Plaza, Kansai Science City, Japan, December 1996.

[59] Leonard N. Foner, “Entertaining Agents: A Sociological Case Study,” *Proceedings of the First International Conference on Autonomous Agents (AA ‘97),* W. Lewis Johnson, ed., Marina del Rey, 1997.

[60] Leonard N. Foner, “What’s an Agent, Anyway? A Sociological Case Study,” *Agents Memo 93-01,*{http://foner.www.media.mit.edu/people/foner/Julia/}

[61] Leonard N. Foner and I. Barry Crabtree, “Multi-Agent Matchmaking,” *BT Technology Journal,* 14 (4), pp. 115+, October 1996.

[62] Leonard N. Foner and Pattie Maes, “Paying Attention to What’s Important: Using Focus of Attention to Improve Unsupervised Learning,” *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB ‘94),* Dave Cliff, Philip Husbands, Jean-Arcady Meyer, and Steward W. Wilson, eds., MIT Press, 1994.

[63] A. Freier, P. Karlton, and P. Kocher, “The SSL Protocol Version 3.0,” IETF draft, March 4, 1996, {ftp://ftp.netscape.com/pub/review/ssl-spec.tar.Z}

[64] Charles Fried, *Privacy* 77 Yale Law Journal 475 (1968).

[65] A. Michael Froomkin, “Flood Control on the Information Ocean: Living With Anonymity, Digital Cash, and Distributed Databases”, 15 U. Pittsburgh Journal of Law and Commerce 395 (1996).

[66] Global Internet Liberty Campaign (GILC) Mixmaster remailer interface, {http://www.gilc.org/speech/anonymous/remailer.html}

[67] Ian Goldberg, “Netscape SSL Implementation Cracked!” message to cypherpunks on September 17, 1995 at 21:41:01 PST.

[68] Li Gong, “An Overview of Enclaves 1.0,” SRI Computer Science Lab, SRI-CSL-96-01.

[69] Li Gong and P. Syverson, “Fail-Stop Protocols: An Approach to Designing Secure Protocols,” *Proceedings of the Fifth IFIP Working Conference on Dependable Computing for Critical Applications*, Dependable Computing and Fault-Tolerant Systems, Urbana-Champaign, pp.44-55, Springer-Verlag, September 1995.

[70] James Gosling and H. McGilton, *The Java Language Environment,* Sun Microsystems, May 1995. {http://java.sun.com/whitePaper/javawhitepaper_1.html}

[71] *Griswald v. Connecticut,* 381 U.S. 479 (1965).

[72] Laura J. Gurak, *Persuasion and Privacy and Cyberspace: The Online Protests over Lotus Marketplace and the Clipper Chip,* Yale University Press, 1997.

[73] Peter Gutmann, “Secure Deletion of Data from Magnetic and Solid-State Memory,” *Proceedings of the Sixth Usenet Security Symposium*, San Jose, CA, July 22-25, 1996.

[74] The Harvest Project, {http://harvest.transarc.com/}

[75] Bennett Haselton, Peacefire, {http://www.peacefire.org/}

[76] Martin Hellman, “DES will be Totally Insecure Within Ten Years,” *IEEE Spectrum*, 16 (7), pp. 32-39, July 1979.

[77] Johan Helsingius, website: {http://www.penet.fi/},
anonymous remailer: {anon.penet.fi},
press release, {http://www.epic.org/privacy/internet/anon_closure.html}

[78] Evan Hendricks, Trudy Hayden, and Jack D. Novik, *Your Right to Privacy, an American Civil Liberties Handbook*, Southern Illinois University Press, 1990.

[79] W. Daniel Hillis, “Co-evolving parasites improve simulated evolution as an optimization procedure,” *Artificial Life II*, Chris Langton, et al, eds., Addison Wesley, 1992.

[80] Lance J. Hoffman, *Building in Big Brother: The Cryptographic Policy Debate*, Springer, 1995.

[81] Douglas Hofstadter, *Fluid Concepts & Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*, Basic Books, 1996.

[82] C. Hornig, “Standard for the Transmission of IP Datagrams over Ethernet Networks,” RFC894, April 1, 1984. {ftp://ftp.isi.edu/in-notes/rfc894.txt}

[83] M. R. Horton and R. Adams, “Standard for Interchange of USENET Messages,” RFC1036, December 1, 1987 {ftp://ftp.isi.edu/in-notes/rfc1036.txt}

[84] B. A. Huberman, ed., *The Ecology of Computation*, North-Holland, 1988.

[85] idealab!, {http://www.free-pc.com/}

[86] I. Ingemarsson and G. J. Simmons, “A Protocol to Set Up Shared Secret Schemes Without the Assistance of a Mutually Trusted Party,” *Advances in Cryptology—EUROCRYPT ‘90 Proceedings*, pp. 266-282, Springer-Verlag, 1991.

[87] *International Traffic in Arms Regulations*, 58 Federal Register 39,280 (1993) (to be codified at 22 C.F.R.§§120-128, 130).

[88] Katherine Isbister and Terre Layton, “Intelligent Agents: A Review of Current Literature,” {http://www.research.microsoft.com/research/ui/persona/isbister.htm}

[89] Aubrey Jaffer and Radey Shouman, SCM, {http://www-swiss.ai.mit.edu/~jaffer/SCM.html}

[90] Richard Jones and Rafael Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, 1996.

[91] Henry Kautz, Al Milewski, and Selman Bart, “Agent Amplified Communication,” *AAAI '95 Spring Symposium Workshop Notes on Information Gathering in Distributed, Heterogeneous Environments*, Stanford, CA.

[92] Bert Kaliski, “Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services,” RFC1424, February 10, 1993,
{ftp://ftp.isi.edu/in-notes/rfc1424.txt}

[93] Alan Kay, “Computer Software,” *Scientific American*, volume 251, number 3, September, 1984.

[94] Kevin Kelly, *Out of Control: The New Biology of Machines, Social Systems, and the Economic World*, Addison-Wesley, 1995.

[95] J. Kephart, "A Biologically Inspired Immune System for Computers," *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, Rod Brooks and Pattie Maes, eds., MIT Press, 1994.

[96] Kenneth E. Kinnear, Jr., ed., *Advances in Genetic Programming*, MIT Press, 1994.

[97] Rob Kling, *Computerization and Controversy: Value Conflicts and Social Choices*, second edition, Academic Press (San Diego), 1996.

[98] Robyn Kozierok, *A Learning Approach to Knowledge Acquisition for Intelligent Interface Agents*, SM Thesis, MIT Department of Electrical Engineering and Computer Science, 1993.

[99] Daniel Kuokka and Larry Harada, "Matchmaking for Information Agents," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) '95*, 1995.

[100] Daniel Kuokka and Larry Harada, "On Using KQML for Matchmaking," *Proceedings of the First International Conference on Multiagent Systems*, Victor Lesser, MIT Press, 1995.

[101] Jaron Lanier, "Agents of Alienation," *interactions*, volume 2, number 3, July 1995.

[102] Chris Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, eds., *Artificial Life II*, Addison-Wesley, 1992.

[103] Yezdi Lashkari, Max Metral, and Maes Pattie, "Collaborative Interface Agents," *Proceedings of the Twelfth National Conference on Artificial Intelligence*, MIT Press, Cambridge, MA, 1994.

[104] Ron Lee, General Counsel, National Security Agency, in an answer to a public question at Computers, Freedom, and Privacy, Cambridge, MA, March 27, 1996.

[105] Tony Lesce, *The Privacy Poachers: How the Government and Big Corporations Gather, Use, and Sell Information about You*, Loompanics Unlimited, Port Townsend, WA, 1992.

[106] Henry Lieberman, "Letizia: An Agent That Assists Web Browsing," *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, Montreal, Canada, August 1995.

[107] Peter Ludlow, ed., *High Noon on the Electronic Frontier: Conceptual Issues in Cyberspace*, MIT Press, 1996.

[108] David Lyon and Elia Zureik, eds., *Computers, Surveillance, and Privacy*, University of Minnesota Press, 1996.

[109] David Mackenzie and Tom Tromey, et al, GNU automake and autoconf, {ftp://ftp.gnu.org/gnu/automake/ and ftp://ftp.gnu.org/gnu/autoconf}

[110] MacroView Communications Corp, {http://www.sixdegrees.com/}.

[111] Thomas Malone and Kevin Crowston, "Toward an Interdisciplinary Theory of Coordination," Center for Coordination Science TR 120, April, 1991.

[112] Pattie Maes, "Agents that Reduce Work and Information Overload," *Communications of the ACM*, volume 37, number 7, ACM Press, 1994.

[113] Pattie Maes, "Intelligent Software," *Scientific American*, volume 273, number 3, pp. 84-86, September 1995.

[114] Pattie Maes and Robyn Kozierok, "Learning Interface Agents," *Proceedings of AAAI'93*, AAAI Press, 1993.

[115] Thomas Malone, Richard Fikes, Kenneth Grand, and Michael Howard, "Enterprise: A Market-like Task Scheduler for Distributed Computing Environments," *The Ecology of Computation*, B. A. Huberman, ed., North-Holland, 1988.

[116] Michael Mauldin, "Chatterbots, TinyMUDs, and the Turing Test: Entering the Loebner Prize Competition," *Proceedings of Twelfth National Conference on Artificial Intelligence* (AAAI-94), Seattle, Washington, August 1994.

[117] Joseph A. Maxwell, *Qualitative Research Design: An Interactive Approach (Applied Social Research Methods Series)*, Sage Publications, 1996.

[118] Tim May, "BlackNet." Private, originally-anonymous communication, later leaked to Usenet newsgroup alt.cypherpunks in 1993 and eventually acknowledged by May to be his.

[119] Robert McChesney, *Telecommunications, Mass Media, & Democracy: The Battle for the Control of U.S. Broadcasting, 1928-1935*, Oxford University Press, 1993.

[120] *McIntyre v. Ohio Elections Comm'n*, 115 S. Ct. 1511 (1995).

[121] J. McQuillan, "Software Checksumming in the IMP and Network Reliability," RFC528 June 20, 1973, {ftp://ftp.isi.edu/in-notes/rfc528.txt}

[122] Matthew B. Miles, and A. Michael Huberman, *Qualitative Data Analysis: An Expanded Sourcebook*, SAGE Publications, Inc., Thousand Oaks, CA, 1994.

[123] George Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller, "Introduction to WordNet: An On-line Lexical Database," Princeton University Technical Report, 1993.

[124] Steven E. Miller, *Civilizing Cyberspace: Policy, Power, and the Information Superhighway*, Addison-Wesley, 1996.

[125] William Mitchell, *City of Bits: Space, Place, and the Infobahn*, MIT Press, 1995.

[126] Janet Murray, *Hamlet on the Holodeck*, The Free Press, Simon & Shuster, 1997.

[127] B. Clifford Neuman and Theodore Ts'o, "Kerberos: An Authentication Service for Computer Networks," *IEEE Communications*, 32(9):33-38, September 1994.

[128] Peter Neumann, mod., *Risks@csl.sri.com* mailing list, {http://www.csl.sri.com/risksinfo.html}

[129] Peter Neumann, *Computer-Related Risks*, Addison-Wesley, 1995.

[130] *Olmstead v. United States*, 277 U.S. 438 (1928).

[131] Marshall Pease, Robert Shostak, Leslie Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM 27/2*, pp.228-234, 1980.

[132] Ithiel de Sola Pool, *Technologies of Freedom*, Harvard University Press, 1983.

[133] M. F. Porter, "An Algorithm for Suffix Stripping," *Program* 14 (3) pp. 130-137, July 1980.

[134] Jon Postel, "User Datagram Protocol," RFC768, August 28, 1980 {ftp://ftp.isi.edu/in-notes/rfc768.txt}

[135] Jon Postel, "Transmission Control Protocol," RFC793, September 1, 1981 {ftp://ftp.isi.edu/in-notes/rfc793.txt}

[136] Jon Postel and Joyce K. Reynolds, "Telnet Protocol Specification," RFC854, May 1, 1983, {ftp://ftp.isi.edu/in-notes/rfc854.txt}

[137] Anton Braun Quist, *Excuse Me, What Was That? Confused Recollections of Things That Didn't Go Exactly Right*, Dilithium Press, 1982.

[138] Thomas S. Ray, "An Approach to the Synthesis of Life," *Artificial Life II*, Chris Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, eds., Addison-Wesley, 1992.

[139] Thomas S. Ray, "A Proposal to Create a Network-Wide Biodiversity Reserve for Digital Organisms," *ATR Technical Report TR-H-133*, {http://www.hip.atr.co.jp/~ray/pubs/reserves/reserves.html}

[140] Recreational Software Advisory Board on the Internet (RSACi) {http://www.rsac.org/}

[141] Michael K. Reiter and Aviel D. Rubin, "Crowds: Anonymity for Web Transactions," ACM TISSEC, June 1998.

[142] Mitchell Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press, 1994.

[143] Doug Riecken, ed., *Communications of the ACM*, special issue on Intelligent Agents, volume 37, number 7, July, 1994.

[144] Howard Rheingold, *Virtual Community: Homesteading on the Electronic Frontier*, Harperperennial Library, 1994.

[145] Howard Rheingold, *Virtual Reality*, Simon & Schuster, 1991.

[146] Bradley Rhodes, "A Continuously Running Automated Information Retrieval System," *AAAI'96 Spring Symposium on Acquisition, Learning and Demonstration: Automating Tasks for Users*, Gil, Birmingham, Cypher, and Pazzani, eds., AAAI Press, 1996.

[147] *Roe v. Wade*, 410 U.S. 113 (1973).

[148] Marc Rotenberg, *The Privacy Law Sourcebook: United States Law, International Law, and Recent Developments*, Electronic Privacy Information Center, Washington, DC, EPIC Publications, 1998.

[149] Marc Rotenberg, *1996 EPIC Cryptography and Privacy Sourcebook: Documents on Wiretapping, Cryptography, the Clipper Chip, Key Escrow, and Export Controls*, Electronic Privacy Information Center, Washington, DC, EPIC Publications, 1996.

[150] Sage Enterprises, Inc., {http://www.planetall.com/}.

[151] Sarah Schafer, "With Capital in Panic, Pizza Deliveries Soar," *The Washington Post*, December 19, 1998, page D1.

[152] Robert W. Scheifler, James Gettys, Al Meno, and Donna Converse, *X Window System: Core and Extension Protocols, X Version 11, Releases 6 and 6.1*, Digital Press, 1997.

[153] Jeffrey I. Schiller, private communication.

[154] Ben Schneiderman, "Looking for the Bright Side of User Interface Agents," *interactions*, ACM Press, January 1995.

[155] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, second edition, John Wiley & Sons, 1996.

[156] Bruce Schneier and David Banisar, *The Electronic Privacy Papers: Documents on the Battle for Privacy in the Age or Surveillance*, Wiley Computer Publishing, 1997.

[157] Philip Selznick, *The Moral Commonwealth*, University of California Press, 1992.

[158] Secure Shell (SSH), {http://www.ssh.fi/}

[159] Upendra Shardanand, *Social Information Filtering for Music Recommendation*, S.M. Thesis, Program in Media Arts and Sciences, 1994.

[160] Upendra Shardanand and Pattie Maes, "Social Information Filtering: Algorithms for Automating 'Word of Mouth,'" *Proceedings of CHI'95*, ACM Press, 1995.

[161] Christoph Schuba, *Address Weaknesses in the Domain Name System*, CS MS Thesis, Purdue University, CSD-TR-94-028, 1994.

[162] Beerud Sheth, *A Learning Approach to Personalized Information Filtering*, S.M. Thesis, MIT Department of Electrical Engineering and Computer Science, 1994.

[163] Beerud Sheth and Pattie Maes, "Evolving Agents for Personalized Information Filtering," *Proceedings of the Ninth Conference on Artificial Intelligence for Applications*, IEEE Computer Society Press, 1993.

[164] Yoav Shoham, "Agent Oriented Programming," *Artificial Intelligence*, volume 60 number 1, pp.51-92, 1993.

[165] Karl Sims, "Evolving 3D Morphology and Behavior by Competition," *Artificial Life IV,* Rodney A. Brooks and Pattie Maes, eds., MIT Press, 1994.

[166] Douglas Smith and Robert Alexander, *Fumbling the Future: How Xerox Invented, Then Ignored the First Personal Computer*, William Morrow, 1988.

[167] H. Jeff Smith, *Managing Privacy: Information, Technology, and Corporate America,* University of North Carolina Press, 1994.

[168] Robert Ellis Smith, *Our Vanishing Privacy,* Loompanics Unlimited, Port Townsend, WA, 1993.

[169] Latanya Sweeney, "Guaranteeing Anonymity when Sharing Medical Data: The Datafly System," *Proceedings of the Journal of the American Medical Informatics Association,* Hanley & Belfus, Inc., Washington, DC, 1997.

[170] Joseph Tardo and Luis Valente, "Mobile Agent Security and Telescript," IEEE CompCon, 1996.

[171] Chelliah Thirunavukkarasu, Tom Fini, and James Mayfield, "Secret Agents—A Security Architecture for the KQML Agent Communication Language," submitted to the CIKM '95 Intelligent Information Agents Workshop, Baltimore, MD, December 1995.

[172] Sherry Turkle, *Life on the Screen: Identity in the Age of the Internet*, Simon & Shuster, 1995.

[173] United States General Accounting Office, *IRS Systems Security and Funding: Employee Browsing Not Being Addressed Effectively and Budget Requests for New Systems Development Not Justified*, GAO report T-AIMD-97-82, April 15, 1997.
PDF: {http://frwebgate.access.gpo.gov/cgi-bin/useftp.cgi?IPaddress=162.140.64.21&filename=gg99001.pdf&directory=/diskb/wais/data/gao}

[174] United States Army CECOM Software Engineering Center, *Rendezvous, Where Mission & Security Meet*, Volume 13, Issue 10, October 1998, {http://www.sed.monmouth.army.mil/114/oct/octnews-r.htm}

[175] United States Federal Privacy Act, 5 U.S.C. §552a (1974).

[176] Vernor Vinge, "True Names," in *True Names... And Other Dangers,* Baen Books, 1987.

[177] W3C Consortium, *Platform for Internet Content Selection (PICS),* {http://www.w3.org/PICS/}

[178] *The Wall Street Journal,* page B1, April 11, 1995.

[179] Andrew Warinner, "That Delivery Boy May Be A Spy," {http://www.xnet.com/~warinner/pizza.html}

[180] Warren and Brandeis, *The Right to Privacy,* 4 Harv. L. Rev. 193 (1890).

[181] *The Washington Times,* August 21, 1991.

[182] Lauren Weinstein, *Privacy-Forum@vortex.com* mailing list, {http://www.vortex.com/privacy.html}

[183] James White, "Telescript Technology: Mobile Agents," *Software Agents*, Jeffrey Bradshaw, ed., AAAI/MIT Press, 1996.

[184] Michael Wiener, "Efficient DES Key Search," *Advances in Cryptology: Proceedings of Crypto '93,* Santa Barbara, CA, Springer-Verlag, 1994.

[185] Harry F. Wolcott, *Writing Up Qualitative Research,* SAGE Publications, Inc., Newbury Park, CA, 1990.

[186] Eric A. Young and Tim Hudson, SSLeay, {http://www.ssleay.org/}

[187] Philip Zimmermann, *The Official PGP User's Guide,* MIT Press, 1995.

[188] Joel Zumoff, "Users Manual for the SMART Information Retrieval System," Cornell Technical Report 71-95.