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

- *Matchmaking* 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).
- *Getting the word out* 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).

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 regulations. 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]:

Toss stopwords

- Very common words (stopwords) are removed.

Toss machine-generated phrases

- Anything matching an *exclusion* regular expression is removed. This gets rid of HTML markup, PGP signature blocks, base64-encoded MIME documents, message 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. Furthermore, 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.