A Framework for Peer-to-Peer Applications

by

Justin Michael Schmidt

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

Massachusetts Institute of Technology

June 2003

Author .....
Justin Michael Schmidt
Department of Electrical Engineering and Computer Science
May 20, 2003

Certified by ....
.....
Barbara H. Liskov
Ford Professor of Engineering
Associate Department Head, Computer Science and Engineering
Thesis Supervisor

Accepted by .....
.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Framework for Peer-to-Peer Applications

by

Justin Michael Schmidt

Submitted to MIT's Department of Electrical Engineering and Computer Science
on May 20, 2003, in partial fulfillment of the requirements for the degree of

Master of Engineering

Abstract

I present the design and implementation of a peer-to-peer application framework. The framework improves user experience and helps developers write better applications. Developers get an open, decentralized, and extensible platform upon which they can build a wide variety of new applications with few technical or political limitations. Users get a platform on which secure, network-aware applications are the norm and run smoothly.

These benefits derive from one simple idea: provide application developers with peer-to-peer primitives that abstract away the underlying networking and security details. The superiority of the resulting applications is evidenced in my implementation of the framework and a few example applications. Application source code is shorter and simpler; applications are developed more easily and more quickly with fewer possibilities for bugs and insecurities.

Thesis Supervisor: Barbara H. Liskov
Title: Ford Professor of Engineering
Associate Department Head, Computer Science and Engineering

Acknowledgments

I would first like to thank my advisor, Professor Barbara Liskov. Her support and expertise helped me focus my ideas into a manageable thesis.

Sameer Ajmani's dedication to improving the framework's design has been invaluable. With his eagerness to read early versions of my thesis, discuss my design decisions and suggest alternatives, and help me find and understand related research, he has been a constant source of both knowledge and motivation over the past year, and I thank him for being a wonderful officemate and mini-advisor.

I would also like to acknowledge the contributions of Karl Magdsick, Jacob Kitzman, Seth Tardiff, Ryan Power, Brian Gilman, and Alison Wong. Alison infused graphical components of the framework's implementation with her artistic talent. Discussions with Brian, Ryan, and Seth helped me recognize the value in designing the framework not just from the developer's perspective, but from the user's perspective as well. Jacob has substantially contributed to the [VFS] application's user interface. Karl deserves special recognition. Since the framework's inception, he has been my sounding board and a faithful supporter. He designed much of the security described in Section 2.3, tirelessly answered my numerous questions, and, above all else, has been a source of inspiration and a true friend. My heartfelt gratitude to you all.

I would like to recognize the many MIT students and professors, especially Professors Patrick Winston, Rodney Brooks, and Ronald Rivest, who have helped me grow and learn over the past six years. The Institute's amazing collection of individuals has instilled in me the belief that I can be the change that I want to see in the world. I wanted technology that did not exist, so I decided to build it. This thesis is the result.

Finally and most importantly, I thank my family for their unconditional love and support. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Problem Introduction

Computers are amazing. They have the ability to manage an enormous amount of mundane detail, perform billions of computations each second, and communicate over thousands of miles instantly and securely. Despite this power, ordinary people have just begun to use their computers to improve their communications with friends, family, and colleagues. Online interaction up to the mid-1990s mainly consisted of email sent and received via server networks. This interaction was well-suited to users' intermittent network access. Network access has become increasingly ubiquitous since then, and online interaction has consisted more and more of direct communication known as instant messaging.

These trends of increasing network availability and increasing online interactions imply that network-aware devices will constantly be at users' fingertips, and virtual interactions will become more and more the norm. Applications running on these devices have the potential for far greater functionality than email or instant messaging. These applications could facilitate new modes of interpersonal communication and could help manage one's personal information and data.

My ultimate ambition is to create a suite of applications that implements these new interactions and functionalities. In this thesis, I present the first steps in the process:

the design and implementation of a framework upon which these applications can easily be built. The framework exploits the inherent similarities in these applications by providing much of the infrastructure needed for networked applications, so that application developers need only focus on functionality and user interface. As a result, application source code can be shorter and less complicated, so that it is developed more easily and more quickly and has fewer possibilities for bugs and insecurities.

1.1 Framework Design Goals

The purpose of the framework is to support the creation of distributed applications, which take the form of plug-ins that extend the functionality of the framework. Clearly, the framework must be *extensible* and support the ability to add these new applications at any time. Similarly, it should be *configurable* and support the ability to remove previously added applications at any time.

Also, the framework must make it easier to write better distributed applications. The distributed application development experience can be a long and arduous process involving many hard subproblems. The framework should simplify application development by solving as many of these subproblems as possible. However, the framework must remain *flexible*; these solutions must not place any limitations on the amount or type of application that can be built.

The framework should address the following difficult subproblems while maintaining complete flexibility:

1. *Security.* Distributed applications manage and exchange significant amounts of private information, so it is desirable to be able to send sensitive data confidentially and reliably as well as trust the legitimacy of the sender and content of received transmissions. Strong guarantees on authenticity, confidentiality, and integrity require strong cryptography, which is a complex and challenging

discipline. The framework provides security primitives in its infrastructure so that applications need not implement their own security scheme.

2. *Decentralization.* On the framework level, no individual node is more important than another. This peer-to-peer nature promotes robust applications that mimic humans' natural method of direct communication while ensuring that the framework itself does not have any central points of control. Applications built upon the framework are self-configuring and self-administering by default, so they are inherently self-sufficient.

   From the application perspective, decentralization is generally desirable, but the client/server model is a well-established, powerful paradigm that may be superior to the peer-to-peer paradigm for certain applications. Despite the decentralized nature of the framework, applications can easily use a centralized service model. Such an application merely treats some nodes differently than others. However, it is important to note that this centralization creates additional possibilities for security and scalability problems as well as introducing the potential for other undesirables into the application, such as censorship, congestion, and dependencies on other people and technologies.

   This last point is especially important from the framework perspective. External developers will be much more likely to build upon the framework if they know they will retain complete control over their applications. If their application's success depends on the benevolence of potential competitors, independent development will be unnecessarily stifled. Complete control over one's application can only be retained if the framework is free from external influences, whether these influences are political or technological. This is only possible with a decentralized, peer-to-peer paradigm.

   In sum, the framework is completely decentralized, yet it supports both peer-to-peer and client/server application paradigms.

3. *Scalability.* Distributed applications should not have unnecessary growth limitations. The framework must be able to incorporate as many users as possible,

and the underlying algorithms must deliver adequate performance, even with millions or billions of users. Communication bandwidth and data storage requirements grow at least linearly with the number of users, so a tremendous amount of data might need to be managed and transmitted. The framework supports massive user scalability via efficient lookup protocols and achieves bandwidth and data scalability through decentralization.

4. *Technology Modularity and Independence.* The framework must not create unnecessary technology dependencies. Wherever possible, applications and users should not be dependent upon any one technology or piece of data, such as a peer-to-peer lookup protocol or an IP address, unless absolutely necessary.

   This philosophy allows the framework to accommodate foreseeable technological developments wherever possible. Applications automatically improve performance as the framework's components are upgraded with advances in the underlying technologies.

5. *Technology Transparency.* The framework must not involve any hidden components or protocols. Hidden components reduce developers' trust and make peer review impossible, and peer review is a crucial component in the process of making secure, robust computer systems.

6. *Popularity.* The popularity of the framework and its applications is of prime importance from both the user and developer perspectives.

   It is clear that the framework's applications must be easy to use and provide desired functionality in order to achieve widespread use. Popularity is important from the user perspective, because a distributed application generally increases its value as its user population increases. For instance, an instant messaging system is better if one can reach more of one's friends. Popularity is also important from the developer perspective, because application development is a time-consuming technical and creative process. It is impossible for any one person to create these applications by oneself; each person is limited in time,

know-how, creativity, and vision. As the developer population increases, application quantity and quality will also increase.

Finally, entrenched applications exist that have much but not all of the functionality I envision. The framework's applications must be clearly superior to have a chance at being competitive and gaining market share. Therefore, the framework encourages independent development; both development teams and individual developers should prefer to develop applications for the framework instead of without it.

1.2 Thesis Overview

In Chapter 2, I detail the design of the framework, including the definition of the framework's application primitives in Section 2.1, an explanation of the primitives' representation in Section 2.2, and a defense of the framework's security in Section 2.3. In Chapter 3, I present my implementation of the framework with a discussion of the application development process, including a detailed listing of implemented proof-of-concept applications in Section 3.1 and descriptions of the user and developer perspectives on the framework in Sections 3.2 and 3.3. I discuss related research in Chapter 4, and I conclude in Chapter 5 with an analysis of my arguments and contributions in Section 5.1, proposals for future work in Section 5.2, and a summary in Section 5.3.

Chapter 2

Framework Design

This chapter describes the design of the framework, namely, how the design goals presented in Section 1.1 are implemented without limiting the number and variety of applications the framework can support. In order to embody these properties, the framework exploits the inherent similarities of distributed applications by providing developers with peer-to-peer primitives. These primitives create an abstraction layer that decouples application development from networking and security complexities. Wherever possible, the framework's design stresses simplicity and intuition. However, some complexity is unavoidable; components of the framework are designed to be as simple as possible, but no simpler.

2.1 Framework Primitives

The framework's abstraction layer provides developers with three peer-to-peer application primitives. These primitives correspond to notions of peer identity, establishing peer presence, and interpeer communication:

1. An *identity* is an unspoofable representation of an entity within the framework.

An identity is synonymous with the terms *principal*, *user*, *node*, and *peer* and simultaneously refers to an instance of the framework on the network and its human user.

2. An identity's presence determines whether that identity is contactable within the framework. An instance of the framework can simultaneously employ many *lookup protocols* to establish presence; lookup protocols translate identities into routable locations such as IP addresses.

3. An identity sends messages to another identity through a *peer channel descriptor*, which is a high-level representation of a channel of communication between identities. The channel has parameters specifying how messages are sent but defaults to strong security.

These primitives supply application developers with an infrastructure that answers the following basic communications questions:

1. With whom we are exchanging data (user abstraction);

2. Where we are sending the data and from where we are receiving responses (location abstraction); and

3. How we are sending and receiving the data (communication abstraction).

Identities represent the concept of *whom*. Lookup protocols map identities to locations within the framework, embodying the concept of *where*. Peer channel descriptors capture the information components which together describe the method (the *how*) of actual communications.

Applications developers use these primitives to represent, locate, and communicate with other peers. The most common way for an application to begin an interpeer connection is to instantiate a peer channel descriptor and tie it to a peer's identity:

```
pcd = new PeerChannelDescriptor(id);
```

When a peer channel descriptor is created, the framework creates a local object representing the channel's properties. By default, data sent via a channel is encrypted, authenticated, reliable, and integrity-checked. In addition, options such as encryption and compression can be independently toggled as desired:

```
pcd.setAuthenticated(false);
pcd.setCompressed(true);
```

Once the desired options have been set, an arbitrary block of data can be sent directly to the target identity:

```
response = pcd.send(data);
```

At this point, the framework uses its default lookup protocols to locate the identity. After successful location, the descriptor sends `data` to `id` in the manner specified by the channel's options. If desired, the channel can be further altered to send anonymously before more data is sent:

```
pcd.setAnonymized(true);
response2 = pcd.send(data2);
```

It may seem peculiar that a channel's options can be changed after the channel is established. This is possible because the sender maintains the channel's state and the receiver does not. Each data item sent re-establishes a connection and re-specifies the transmission options. The overhead of this connection re-establishment is drastically reduced via a low-level Kerberos-like ticketing optimization [6] explained in Appendix A.

If the developer wishes to specify a preference for the set of lookup protocols, an ordered list of lookup protocols can be specified:

```
pcd2 = new PeerChannelDescriptor(id2);
pcd2.addLookup("Ring");
pcd2.addLookup("Blob");
```

This code specifies that the developer wishes to use the "Ring" lookup to locate `id2` first, and if that fails, to use the "Blob" lookup next. Finally, if neither succeeds, possibly because these lookups are not installed, the framework will use any other of its default lookups to try to locate `id2`. Ring and Blob are two implementations of lookup protocols discussed in more detail in Section 2.2.2.

Applications with strict communication requirements, such as a strong anonymity guarantee, may wish to maintain stringent control over the lookup process. Such an application may choose to use only specific lookup protocols and not fall back on the framework's default lookups. This can also be specified:

```
pcd3 = new PeerChannelDescriptor(id3);
pcd3.addLookup("Blob");
pcd3.setLookupDefaultsMode(false);
```

This channel will attempt to locate `id3` by using only the Blob lookup. If Blob is unable to find `id3` or Blob is not installed, the lookup will fail. This option is not recommended for general use because such an application has a brittle dependency on a lookup technology. This application will fail if the Blob lookup is not installed, and applications that use such a peer channel descriptor will not migrate to new (default) lookups that might be developed. It is important to note that, even though such an application breaks forward compatibility, it does so in an intentional, opt-out fashion.

There are further options for many parts of this process. The full peer channel descriptor API can be found in Appendix B.

2.2 Primitive Representation

The framework represents the three fundamental primitives of identity, lookup protocols, and peer channel descriptors in a manner that stresses flexibility and simplicity.

2.2.1 Identity

Identity is the term for an entity within the framework. Each identity is represented by a unique 128-bit number, which is tied to cryptographically strong authentication in the form of a private key. Upon joining the framework, a new entity creates a public/private key pair. The entity's identity is the first 128 bits of the SHA-1 hash of the public key.

Since only the new entity knows the corresponding private key, only messages that are signed with its private key appear to come from its identity. Thus, identities within the framework are unspoofable under standard hardness assumptions. This public key-based identity system is self-verifying and decentralized, yet is not a traditional public key infrastructure. Since identities are also self-creating, Sybil attacks [5] are feasible. This concern is addressed in Section 2.3.

It is worth noting that asymmetric keys are the only acceptable method of strong authentication. MAC addresses, IP addresses, and GPS locations are all spoofable [13]. Passwords or other secrets that must be shared to be verified require centralization or unacceptable levels of trust in the other party.

2.2.2 Lookup Protocols

The framework can use multiple, simultaneous lookup protocols to establish an identity's location and availability within the framework. Normally, each lookup protocol shares the same identity space; that is, a user's identity does not change from

one lookup protocol to another. If absolutely necessary, however, a lookup protocol may use its own internal identifier scheme. The process of creating an independent identifier scheme is problematic and not recommended for security reasons that are discussed in Section 2.3.

The framework has two standard lookup protocols:

1. *Ring.* For its primary lookup protocol, the framework uses a Chord-like, structured network topology that provides logarithmic scalability and robustness in the face of frequent node presence fluctuation [11]. The underlying protocol remains intact, save for a minor augmentation necessary for flexibility and security purposes; instead of Chord's (spoofable) IP-based identifier system, framework identities are used. This change is critical to the goal of strong authentication; IP-based authentication is insufficient [13]. Also, location-based identification schemes are inflexible, whereas an identity scheme based on a public key is portable. Requiring identities to have unchanging locations is an unnecessary limitation.

2. *Blob.* In addition to running a Chord-like ring, the framework independently runs a Freenet-like unstructured network topology [3]. The routing algorithm is modified to allow routing by identity, and the routing table is partitioned into two equal halves called *generations.* Entries in the *young* generation of the routing table are updated using LRU replacement, just as in Freenet. Entries in the *old* generation are only replaced when periodic contact attempts have been unsuccessful for several hours. The matching function employs a "generation gap" heuristic, ranking the best matches in the older generation above all entries in the younger generation. The number of hours for contact timeout and the number of matches in the older generation before "bridging the gap" are configurable parameters set to three and five, respectively.

The primary motivation for allowing simultaneous dissimilar lookup protocols in the

framework design is based upon applications' inherently different needs. Applications are likely to have a preference for the strengths inherent to a particular lookup protocol, such as performance, robustness, or security. The Ring lookup is more efficient and reliable under normal operation, but it is not well-suited to all possible applications. For instance, it does not attempt to provide anonymity like the Blob lookup. Similarly, there are tradeoffs in overlay vulnerabilities; the Ring topology is more vulnerable to attacks by colluding malicious nodes, but these weaknesses are well-complemented by the Blob topology, which is more resistant to these types of attack. Multiple protocols also make the framework less brittle; several simultaneous failures must occur for abject lookup failure. Similarly, multiple protocols also ease application migration to new protocols, allowing users and developers to experiment without sacrificing the reliability of more proven protocols.

2.2.3 Peer Channel Descriptors

Peer channel descriptors handle the process and details of sending data from one identity to another. The data transfer process involves several steps: serializing application data into a *metagram*, securely transferring the metagram according to the channel's parameters via the *point-to-point protocol*, and *dispatching* received metagrams to the appropriate application.

Metagrams Metagrams are the framework-level representation of data and metadata and are designed to accommodate almost any type of data. Such flexibility can only be accomplished through simplicity; as such, a metagram consists of two fundamental parts: its *label* and its *data*. The label field is a tag for the type of the data, and the data is an arbitrarily long sequence of bits. One example of a metagram is a simple representation of an instant message, where the label would be "IM" and the data would be the message "Hey, are you there?". When displayed in a human-readable format, such a metagram looks like:

```
[Schedule]
  [Invitation]
    [EventName] PMG group meeting
    [Invitees]
      [id:Guest] 0214A43F54FD5AFC490AB15287CE1466
      [id:Guest] 1E1E0D08B642753777A499AB4F6C7CEA
      ...
    [s64i:StartTime] 267F309EAB28100E
    [s64i:EndTime] 267F309EAB5EFE8E
    ...
```

Figure 2-1: *A metagram example of multi-dimensional data representation.*

```
[IM] Hey, are you there?
```

However, this two-part metagram representation is not yet sufficiently general. The ability to nest metagrams is crucial, because not all data is one-dimensional. For example, an invitation to an event could be structured as in Figure 2-1.

The ability to create arbitrary-dimension metagrams allows arbitrarily complex data to be modeled. This requires but one change to the metagram representation: adding a *container bit* that is flipped as needed to indicate a *container metagram*, which is a metagram whose data represents one or more internal metagrams. `[Invitation]` and `[Invitees]` are examples of container metagrams, while `[s64i:EndTime]` and `[id:Guest]` are examples of *non-container metagrams*, i.e. metagrams whose data is not an encoding of internal metagrams. Note the optional type specification in the label of non-container metagrams; `id` suggests that the data of each `Guest` metagram be interpreted as a framework identity, and `s64i` suggests that the data of the `StartTime` and `EndTime` metagrams be interpreted as signed 64-bit integers. Labels without a suggested type are interpreted as UTF-8 encoded strings.

Container?	*Label Length*	*Data Length*	*Label*	*Data*
1 bit	15 bits	32 bits	x bytes	y bytes

Table 2.1: *Metagram bitwise representation.*

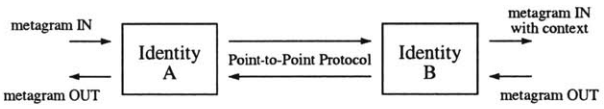

Figure 2-2: *An illustration of the point-to-point protocol.*

The definition of a metagram is now complete. A metagram is serialized as in Table 2.1, where x, the length of the UTF-8 encoding of the label, is limited to at most $2^{15} - 1$ bytes, and y, the length of the data, is limited to at most $2^{32} - 1$ bytes.

Point-to-Point Protocol Metagrams are sent securely between identities via the point-to-point protocol. This protocol is based on the public key "infrastructure" established in Section 2.2.1, whereby identities can trust each others' public keys because one's public key must hash to one's identity, and, with high probability, it is impossible to find another public key that also hashes to the same identity. This communication abstraction is illustrated in Figure 2-2. It is important to note that once a metagram is received, the context of that metagram is immediately attached to it, and a *context metagram* is formed. The context includes the identity of the sender and information about how to return a response metagram.

The cryptographic details of the protocol are similar to existing point-to-point protocols [4]; description and brief analysis of the protocol is included in Appendix A.

Metagram Dispatch After the content metagram is repackaged with the identity of the sender into a *context metagram*, the context metagram is given to the local *metagram dispatcher*. The dispatcher looks up the label of the internal content metagram in its application hashtable and hands the entire context metagram off to the appropriate *application handler* for application-specific processing. This process is graphically represented in Figure 2-3.

Figure 2-3: *An illustration of the process of context metagram dispatch.*

2.3 Framework Security

In order to discuss the security of the framework, it is necessary to address two orthogonal concerns: *point-to-point security* and *overlay security.*

2.3.1 Point-to-Point Security

Much like any other distributed system, the framework involves point-to-point communication. Fortunately, the point-to-point communication paradigm is well understood [4]. Unfortunately, many of the peer-to-peer applications familiar to the public are lacking in fundamental security measures. The framework provides the application developer with simple attribute customization for encryption, authentication, reliability, integrity-checking, anonymity, and compression. Applications default to maximum levels of point-to-point security unless otherwise specified.

However, the user has ultimate control of how data is sent; one can globally override application settings for channel parameters. For instance, the user could force all data sent via a given application to be compressed and encrypted. If the World Wide Web were designed in this manner, this power would be analogous to the ability to turn on SSL/TLS [21] encrypted web browsing for any website, regardless of what the webmaster or browser architect had in mind.

2.3.2 Overlay Security

While point-to-point security is well understood, peer-to-peer overlay security is relatively new and quite challenging, which is a dangerous combination for applications that require strong security guarantees. Solutions to many overlay-level security problems, such as denial-of-service via a Sybil attack [5], remain elusive. While overlay security is somewhat beyond the scope of this discussion, the framework nonetheless

is designed to mitigate known attacks as much as possible.

There are several important issues to consider:

1. Unspoofable identities are critical for overlay security. If identities are not based upon strong authentication in one way or another, point-to-point security cannot be achieved, and overlay security becomes moot.

2. Entities must not be able to control a large number of identities. If they are able to do so, then devastating denial-of-service attacks can be engineered against an overlay. The only defense against such a Sybil attack requires identity certification [5]. When an identity wishes to join the framework, it presents a *real world identifier* such as a credit card number or an IP address and requests a certificate from a certificate authority (CA). The CA must decide whether the join request is acceptable based on the previous requests associated with the real world identifier. Only identities with certificates are trusted to be routing table entries, which means identities without certificates will not be locatable.

   The choice of certifying a request is a delicate one and could probably be a separate thesis topic in and of itself. Nonetheless, the framework's approach to this problem is to have a joining node present a real world identifier. Subsequent joins on the same resource beyond a given threshold would incur exponentially increasing costs, such as non-parallelizable computation [8] for an IP address or currency for a credit card number, or even would be rejected in extreme cases. If an overlay already has a reasonably large number of non-malicious nodes, a dedicated attacker would need to accumulate a large number of credit cards and/or an enormous amount of currency, or, in the case of IP addresses, have access to a large, sparse IP address space and/or an extremely large non-parallelizable amount of computation.

   It is important to note that denial-of-service attacks are not limited to peer-to-peer networks. If an attacker had access to the resources listed above, similarly effective attacks can be generated by more "traditional" means [31]. Even if

the peer-to-peer denial-of-service issue were to be resolved, the denial-of-service vulnerability remains through other avenues for attack.

3. Entities must not be able to choose their own identities [3,5,10]. For many lookup protocols, including the Ring structured lookup, if entities could choose their own identities, attackers would be able to cluster around specific victims and partition them from the rest of the network [1]. Essentially, this means that identities must be provably random. This is easily accomplished as part of the certificate granting process. The entity sends its public key to the CA. and the CA twiddles the public key and signs the resulting public key. The signed public key and information about how to generate the corresponding private key are sent back to the entity with its certificate.[1] Since the identity is the hash of the new public key, the identity is randomized. Also, it is important to note that no one (not even the CA) learns any information about the entity's private key.

It is important to note that, given a CA, many other authentication systems are reasonable. For instance, the CA could assign an entity a random number whose hash would be its identity. The identity would be correlated to the public key in the certificate instead of correlated by hashing the public key. This would even allow existing public keys to be incorporated within the framework.

However, I believe that incorporation of a CA into a peer-to-peer system somewhat violates the ideal of pure decentralization. I have demonstrated the need for a CA, but I assert that CAs should be given as little responsibility as possible. The framework relies on the CA only to mitigate known overlay denial-of-service threats. If a new lookup protocol were developed that did not have these vulnerabilities, the framework would have no need for the CA and would be completely decentralized. Any overlay

[1]This type of key modification is straightforward with RSA, DSA, and ElGamal keys. The implementation uses DSA and works as follows: x is one's secret key parameter, and the public key is $\langle p, q, g, g^x \bmod p \rangle$. The CA generates a $k \in \{0,1\}^{64}$ and signs and returns the new public key, $\langle p, q, g, g^{x+k} \bmod p \rangle$, and the twiddle parameter, k. One's new private key is $x' = x + k$. Note that $1 < x <= q - 2^{64}$ must originally be true.

security system that gives the CA more responsibility than this will never be able to achieve true decentralization.

Chapter 3

Framework Implementation

In an effort to substantiate my claims of the framework's flexibility and extensibility, I implemented the framework design using about 10,000 lines of Java code. Throughout the following discussion, I use the terms *application* and *service* in two different senses. I use the term *application* in the sense of a substantial, cohesive software package. Applications would be independently insertable into the framework. Some applications may export *services* that can be used by other applications. For instance, the [Contact] application exports functionality that translates identities into human-readable aliases, and this functionality can be incorporated into other applications.

It is important to note that some minor details of the design are not yet implemented; I list these omissions in Section 3.4.

3.1 Implemented Applications

I have built several example applications based upon the framework's implementation. Although there is too much detail for thorough explanation here, I demonstrate proof-of-concept for the framework's capabilities by highlighting the noteworthy details

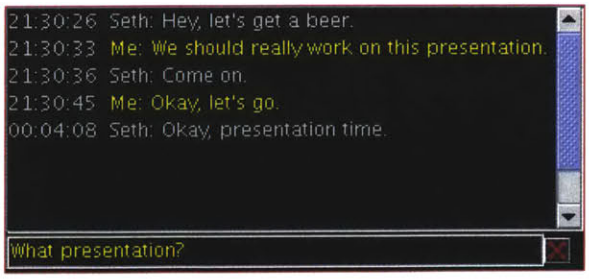

Figure 3-1: *An* ⌊IM⌉ *window with a conversation.*

about the applications' implementation.

1. **Instant Messaging** – ⌊IM⌉

   Instant messaging (IM) is an application in which two users converse by sending each other short text messages. ⌊IM⌉ is an implementation of a simple instant messaging application that is completely secure and decentralized, yet it constitutes only 25 lines of handler code and 200 lines of GUI code. The `IMHandler` code is attached in Appendix B; see Figure 3-1 for a screenshot of an ⌊IM⌉ conversation window.

2. **File Transfer** – ⌊File⌉

   I implemented a simple, secure FTP-like application that can send and receive individual files. This application uses about 225 lines of code, and the process of creating this application is thoroughly described in the developer's walkthrough in Section 3.3.

3. **Lookup Protocols** – ⌊Ring⌉, ⌊Blob⌉

   As a testament to the flexibility of the framework's application model, the lookup protocols as described in 2.2.2 are written and interface with the framework just like any other application. ⌊Ring⌉ is the implementation of the structured, Chord-like topology, while ⌊Blob⌉ is the implementation of the unstructured, Freenet-like topology. Each is completely secure and involves about 1,000 lines of code.

```
[WAnonymity]
  [Payload]
    [Content]
      ...
    ...
  [id:Destination] ...
  [ba:Nonce] ...
```

Figure 3-2: *An example* `[WAnonymity]` *metagram.*

The only fundamental difference between lookup applications and non-lookups is that each lookup application also registers itself as a lookup protocol for the framework. This is not fundamentally different from other applications, which also plug-in (even at runtime) merely by registering with the framework to receive their types of metagrams. ⌊Ring⌉ and ⌊Blob⌉ just register as a lookup protocol at the same time. Also, the lookup protocols use different peer channel descriptor constructors than those described in Section 2.1; namely, they use constructors that use an IP address or DNS name and TCP port. These constructors are described in Table B.1 in the peer channel descriptor API in Appendix B.

4. **Weak Anonymity** – ⌊WAnonymity⌉

   ⌊WAnonymity⌉ provides an anonymization service for other applications. Any metagram from any application can be sent in a weakly anonymous fashion. The content metagram is wrapped into a `[WAnonymity]` metagram with the metagram's ultimate destination and a random nonce as represented in Figure 3-2.

   The originator of the anonymous metagram starts by sending the `[WAnonymity]` metagram off to a random identity. That identity receives the metagram via the `WAnonymityHandler`, which looks to see if its own identity is the intended destination and, if so, processes the payload metagram and deposits its embedded content into the metagram dispatcher. On the other hand, if its own identity is not the intended destination, the handler modifies the nonce of the

[WAnonymity] metagram. With a probability p, the handler delivers the metagram to the intended recipient; with probability $1 - p$, the handler chooses yet another random node and passes the metagram along, keeping track of the forward and backward nonces and the sender for a short while so that a response can be returned back along the same path. In the implementation, p is a configurable parameter set to 0.1.

5. **User Notification** – ⌊Notification⌉

   The user notification application maintains a list of events and/or occurrences that warrant requesting the human's attention. Each event/occurrence is embodied in a [Notification] metagram, which provides supporting details, such as its description, its source, its urgency, and a timestamp.

   ⌊Notification⌉ is a simple example of an application that provides a service for other applications. That is, other applications need not implement their own user notification scheme; instead, they can just place [Notification] metagrams into the metagram dispatcher. Note that ⌊Notification⌉ does not have a networked component; NotificationHandler ignores notifications not from one's own identity.

6. **Contact List** – ⌊Contact⌉

   Much like a traditional address book application, the ⌊Contact⌉ application manages information about users, such as their name, email address, phone numbers, and the like. Only two fields deserve special mention: *identity* and *alias*. The first corresponds to the contact's 128-bit identity within the framework, and the second is a unique handle by which the contact can be referred to (locally) by other applications. The second is a more user-friendly format than a 128-bit number.

   ⌊Contact⌉ is far superior to a standard address book application for two main reasons. First, it provides valuable services that other applications can use, most notably the ability to translate between an alias and an identity. Note in

the ⌊IM⌉ window in Figure 3-1 that I appear to be having an conversation with "Seth", not a 128-bit number. Another valuable service is group management, whereby collections of contacts can be treated as a single entity throughout the framework.

Second, the framework's networked nature creates potential for additional functionality. ⌊Contact⌉ detects who is online and acts as a focal point for interactions with any identity. Perhaps more mundanely, users can acquire each other's contact information easily. Users can request another's contact information, or one can send one's own information directly to someone else, much like a virtual business card.

7. **To-Do List** – ⌊Task⌉

   ⌊Task⌉ is a simple application that helps one manage a to-do list. To-do items are stored as `[Task]` metagrams that contain standard fields for the task's category, description, status, priority, and due date. `[Task]` metagrams also contain three fields that require some explanation: `[Task-ID]`, `[Partners]`, and `[Updates]`. `[Task-ID]` represents an identifier for a task. This is required because to-do items in ⌊Task⌉ can have dependencies on other to-do items and also can be shared with other identities, namely the identities listed in the `[Partners]` metagram. Any changes to a shared task send `[Update]` metagrams to each of the identities in `[Partners]`. Each received `[Update]` metagram is cataloged in the `[Updates]` metagram, providing an audit trail for the changes. Much of the ⌊Task⌉ application is exemplified in the dialog for creating and editing to-do items; Figure 3-3 is a screenshot of this dialog.

8. **Calendar and Appointments** – ⌊Schedule⌉

   ⌊Schedule⌉ is an integrated calendar in which users can plan their daily schedule and manage their time commitments. Unlike more traditional scheduling and productivity software, users can make details about their calendar readable (or even modifiable) by select individuals and can send each other invitations

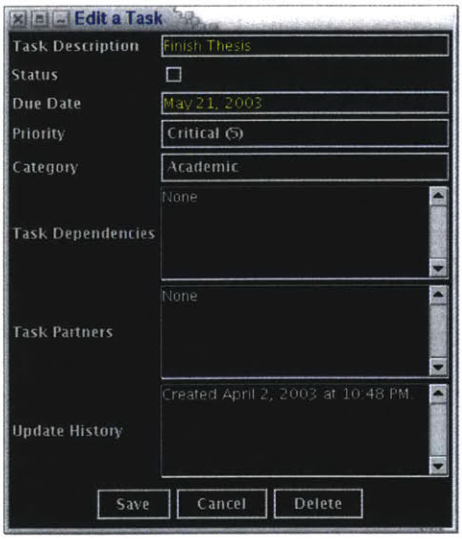

Figure 3-3: *A* ⌊Task⌉ *window for editing a to-do item.*

to meetings or parties. These invitations integrate with one's schedule and automatically update as invitees respond.

9. **Email Client** – ⌊Email⌉

   ⌊Email⌉ is an excellent example of how a traditional application can be greatly augmented by running within the framework. Each user in ⌊Contact⌉ has an *email address* field. ⌊Email⌉ can be configured to send emails that are destined for addresses that are contained in ⌊Contact⌉ through the framework instead of through the normal outgoing email server. This provides two notable benefits over the traditional route: secure email and elimination of email size limitations.

   ⌊Email⌉ also addresses one major drawback of the framework's serverless design: the unavailability of users when offline. What is one to do if one wants to send a metagram to an offline user and then go offline oneself? A simple solution to this problem is to bootstrap the framework on top of users' traditional email.[1] That is, if one is offline, one can still be sent metagrams as attachments in a standard email.

   Upon checking one's traditional email with an ordinary email client, these meta-

[1] A more complicated solution to this problem is to store the metagram within the framework itself. I discuss my concerns about framework-level data storage in Section 5.2.

grams will be downloaded from the email server and then must be manually inserted into the framework for processing. Using [Email] as one's email client makes this process much nicer. Any metagrams received while offline are automatically processed upon download of such emails.

It is also interesting to note that [Email] is an application that communicates with established services over traditional protocols such as IMAP4 and POP3 as well as interacting over the framework.

10. **Virtual File System** – [VFS]

[VFS] is an application that allows users to map files on their machines to virtual directories in a virtual file system. Users can share the data in these virtual directories by setting directory-specific access rights via access control lists (ACLs) much like AFS [28]. [VFS] can be thought of as web serving with ACLs and strong authentication, and it is an example of a server-like application that provides functionality that will primarily be used by identities other than oneself.

Perhaps most importantly, [VFS] provides powerful services upon which other applications can build: a distributed file system for the framework.

11. **Automatic Backups and Mirroring** – [Sync]

Frequently, some of the non-optimal aspects of the framework can be drastically improved by adding an application. For example, lookups solve the connectivity problem, [Contact] solves the problem of user-unfriendly 128-bit identities, and [Email] addresses the unavailability of users while offline. Now, we solve the ordinarily non-trivial problem (for servers) of keeping reliable backups of user data by building upon [VFS].

Essentially, backups only require the ability to write to a directory in another identity's [VFS] share. For example, if I have `karlm` aliased to an identity in [Contact], scripts such as "Encrypt and save all of `~/.is/` to `karlm:/jbackup/` on an hourly basis", which backs up all framework data to `karlm`'s `/jbackup/`

directory, are easy to create and automate.

Mirroring is similar to backup and is useful for synchronizing a PDA or maintaining a local copy of a friend's network share. The "master" copy of ⌊VFS⌉ data can be monitored by any number of clients, which always update their state to reflect the current state of the master copy. This can be thought of as a "pull" backup rather than a "push" backup, and content to be mirrored must already be accessible on ⌊VFS⌉.

The simplicity of these example applications highlights the framework's ease of extensibility and adaptability to a variety of applications. However, these applications by no means represent the gamut of applications that are possible. Essentially, the only current limitation on the types of applications that can be built using the framework is one's imagination.

3.2 User Perspective

Applications fulfill a variety of users' needs: quick communication, sharing of schedules, sharing of files, and online collaboration with friends, family, and colleagues. The framework's single authentication mechanism eliminates the need for a username and password for each application, and the integration of implemented application functionality gives the software a smooth feel, despite the fact that its implementation has progressed only to the extent of beta-level code (at best).

Figure 3-4 shows a screenshot of a main view of the framework. GUI components of the ⌊Contact⌉, ⌊Notification⌉, and ⌊Schedule⌉ applications are displayed from top to bottom, followed by the installed applications tray. The ⌊Contact⌉ component displays which of my contacts are online and integrates easy access to user-level and group-level actions of other installed applications. Note ⌊Contact⌉'s embodiment of a group representation for collections of identities, such as the group "IS-Dev". The

Figure 3-4: *A screenshot of the framework displaying my contacts (top), my notifications (middle), and the application tray and current month's calendar (bottom).*

[Notification] component deals with user attention management; in this screenshot, nothing requires the user's attention. The [Schedule] component displays the calendar for the current month, and, upon clicking on a date, the application will jump to one's schedule for that day.

3.3 Developer Perspective

One of the original design goals of the framework was to allow developers to create new applications easily. In this section, I explain the process of writing a simple application, namely the [File] application previously mentioned in Section 3.1, and I show that the development process is greatly simplified by the framework's application

building blocks.

3.3.1 ⌊File⌉ Application Design

I wrote the ⌊File⌉ application to enable users to send any number of files directly to one another in a secure manner. This desired functionality requires two main behaviors from a ⌊File⌉ application instance: sending and receiving files. The framework makes the transmission of files across the network quite simple, as files are encapsulated within metagrams and sent via the metagram exchange methods previously discussed in Section 2.2.3. The ⌊File⌉ application uses peer channel descriptors to transmit all of the application's data, and the rest of the application's code allows the sender to choose files to send and allows the receiver to choose which files to receive.

The sender initiates file transfer by choosing files to be sent and the desired receiver and transmitting to the receiver a *transfer request*, which provides key information about the files that the sender wishes to transmit, namely the file names and sizes. The receiver processes the transfer request before returning a *request response* that indicates which files from the original request that the receiver accepts and rejects. The sender processes the request response, and accepted files are then transferred sequentially.

These steps are encoded in the ⌊File⌉ protocol, whose standard network communication is illustrated in Figure 3-5 and detailed in Sections 3.3.2 through 3.3.5. Complete pseudocode for the application design is presented in Section 3.3.6. A short discussion of how the application interfaces with the framework's implementation is presented in Section 3.3.7, and I summarize the developer's benefits in Section 3.3.8.

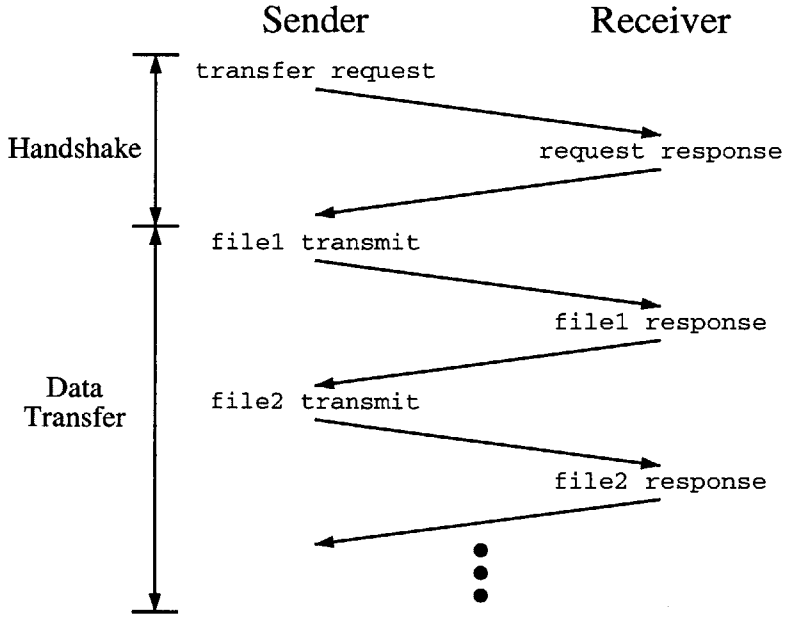

Figure 3-5: *Anatomy of the* [File] *application's network communication.*

3.3.2 Constructing a Transfer Request

The sender initiates file transfer by transmitting a transfer request to the desired target. A transfer request conveys that the sender wishes to send files to the receiver and includes a summary of the files to be transferred. This request is embodied in a `[TransferRequest]` container metagram, and the sender places `[SendRequest]` metagrams inside the `[TransferRequest]` metagram for each file to be sent. Each `[SendRequest]` metagram contains the information for one file, including the file's name and size. The overall request is wrapped inside a `[File]` metagram so that the framework instance running on the receiver's end knows to give the metagram to the [File] application. The anatomy of a transfer request is embodied in the example `[TransferRequest]` metagram in Figure 3-6.

3.3.3 Responding to a Transfer Request

Upon receipt of a transfer request by the receiver's `FileHandler`, the handler processes it and presents the metagram's request to the user. The user chooses which files

```
[File]
  [TransferRequest]
    [SendRequest]
      [ba:FileID] 4F6A8E103583AB07
      [FileName] 6.829.ppt
      [s64i:FileSize] 113664
      ...
    [SendRequest]
      ...
    ...
```

Figure 3-6: *A* `[File]` *metagram describing a* ⌊File⌉ *transfer request.*

to accept and which to reject, and the application keeps track of accepted `FileIDs` in an *accepted files hashtable* and sends back a `[RequestResponse]` metagram containing internal `[ba:AcceptID]` metagrams, where the data for each `[ba:AcceptID]` metagram corresponds to the data from the `[ba:FileID]` metagram in the original request. In the instance where a file is rejected instead of accepted, a `[ba:RejectID]` metagram is returned instead, and the `FileID` is not added to the hashtable. An example `[RequestResponse]` metagram is provided in Figure 3-7.

3.3.4 Transmitting File Data

The sender reads in the `[RequestResponse]` from the receiver and, for each `FileID` that was accepted, constructs a `[FileData]` metagram that includes the `FileID` field and the data of the file. The `[FileData]` metagrams such as the one listed in Figure 3-8 are inserted into new `[File]` metagrams and sent to the receiver.

```
[File]
  [RequestResponse]
    [ba:AcceptID] 4F6A8E103583AB07
    [ba:AcceptID] ...
    [ba:RejectID] ...
    ...
```

Figure 3-7: *A* `[File]` *metagram describing a* ⌊File⌉ *request response.*

```
[File]
  [FileData]
    [ba:FileID] 4F6A8E103583AB07
    [ba:Data] ...
  [FileData]
    ...
  ...
```

Figure 3-8: *A* `[File]` *metagram representing file data.*

3.3.5 Receiving and Responding to File Data

The `[FileData]` metagrams arrive through the `FileHandler` just like the transfer requests. The handler checks to see if the `FileID` of the `[FileData]` metagram is in the accepted files hashtable, and if so, saves the file with the previously specified filename in the appropriate location and responds with a `[ba:ACK]` metagram. If the `FileID` does not match anything in the accepted files hashtable or some other error occurs, a `[ba:ERR]` metagram is returned instead, signifying rejection of the file. The two possible responses are illustrated in Figure 3-9.

3.3.6 ⌊File⌉ Pseudocode

In order to make the conciseness of the ⌊File⌉ application's as clear as possible, I have included pseudocode for the functionality presented in this section. This pseudocode comes in two major components: a `SendFileAction` class and a `FileHandler` class. The `SendFileAction` class embodies the functionality involved in sending a file, including constructing `[TransferRequest]` and `[FileData]` metagrams, and is listed in Figure 3-10. The `FileHandler` class manages ⌊File⌉'s network traffic from

```
[File]
  [ba:ACK] 4F6A8E103583AB07
```

```
[File]
  [ba:ERR] 4F6A8E103583AB07
```

Figure 3-9: *Metagrams for positive and negative responses to incoming file data.*

the receiver perspective, including processing of [TransferRequest] metagrams and construction of [RequestResponse] metagrams, and is listed in Figure 3-11.

The SendFileAction class embodies a service that ⌊File⌉ exports: sending files. This action can be instantiated and executed anywhere, so it can be incorporated into any other application.

3.3.7 Framework GUI Interface

It is worth noting that the application's user interface extends naturally from the protocol's network communication. The code for the class that represents the ⌊File⌉ application is listed in Figure 3-12. It shows how simply the application interfaces with the rest of the framework. This interface mainly consists of applications exporting their actions, namely instances of ApplicationAction subclasses. ⌊File⌉ exports the SendFileAction as a "buddy action", which means the SendFileAction will be available whenever the framework gives the local user a choice of single-user actions. One example of this is right-clicking on another identity in the contact list as shown in Figure 3-4. One can see that the SendFileAction is incorporated in the list of available actions displayed in the framework's GUI. If desired, the ⌊File⌉ application could also export multiple-user actions via a getGroupActions method, or menu actions via a getMenuActions method.

3.3.8 Developer Perspective Summary

The ⌊File⌉ application has been described using only about 80 lines of pseudocode. The code's simple and succinct nature is a direct result of the power of the framework's communication abstraction; each instance of data transfer takes one line of code. In the context of the entire pseudocode, the identity and peer channel descriptor primitives are overshadowed by code that is predominantly related to user interfacing and

```
class SendFileAction extends ApplicationAction {
  private Identity idReceiver = null;
  //  create a new action that will send a file to the given identity
  public constructor SendFileAction(Identity id) {
    super("Send File to"); idReceiver = id; }
  //  execute the send file process to identity "idReceiver"
  public void execute() {
    file-list = pop up GUI for user to choose files to send;
    //  create a hashtable to keep track of send requests
    Hashtable htFiles = new Hashtable();
    //  create empty container metagram named "File"
    Metagram mg = new Metagram("File", true);
    Metagram mgTR = new Metagram("TransferRequest", true);
    //  add send request metagrams for each file
    for each file in file-list {
      Metagram mgSR = new Metagram("SendRequest", true);
      //  get eight random bytes as the file identifier
      byte[] file-ID = FrameworkUtils.getRandomBytes(8);
      htFiles.put(file-ID, file);
      //  fill in the fields of the send request
      mgSR.add(new Metagram("ba:FileID", file-ID));
      mgSR.add(new Metagram("FileName", file.getName()));
      mgSR.add(new Metagram("s64i:FileSize", file.getSize()));
      mgTR.add(mgSR); // add SendRequest to the TransferRequest
    }
    mg.add(mgTR); // add TransferRequest to File metagram
    //  create channel to the receiver
    PeerChannelDescriptor pcd = new PeerChannelDescriptor(idReceiver);
    //  send transfer request and wait for response
    Metagram mgResp = pcd.send(mg);
    //  process response for accepted files
    accepted = mgResp.get("RequestResponse").getAll("ba:AcceptID");
    for each file-ID in accepted { // construct FileData metagrams
      mg = new Metagram("File", true);
      Metagram mgFD = new Metagram("FileData", true);
      mgFD.add(new Metagram("ba:FileID", file-ID));
      mgFD.add(new Metagram("ba:Data", htFiles.get(file-ID)));
      mg.add(mgFD); // add FileData to File metagram
      if(pcd.send(mg).get("ba:ACK") is file-ID) notify success;
      else notify the user that file was rejected by receiver;
    } } }
```

Figure 3-10: *Pseudocode for sending a file with the* [File] *application.*

```
class FileHandler extends ApplicationHandler {
  public constructor FileHandler() { super("File"); }
  //  processes "File" context metagrams from the metagram dispatcher
  public Metagram handle(ContextMetagram cm) {
    check cm for proper format;
    Identity idSender = cm.getSender();
    Metagram mg = cm.getContentMetagram();
    //  process transfer requests
    if(mg.get("File").get("TransferRequest") is not null) {
      requests = get send requests from transfer request;
      accepted-requests = pop up GUI of requests;
      create RequestResponse metagram mgRR;
      for each accepted request in accepted-requests {
        add file-ID to hashtable for accepted files;
        create "ba:AcceptID" metagram and add to mgRR;
      }
      add "ba:RejectID" metagrams for each rejected file;
      add mgRR to "File" metagram and return "File" metagram;
    }
    //  process file data
    if(mg.get("File").get("FileData") is not null) {
      get file-ID from "FileData" metagram;
      if(accepted files hashtable has file-ID) save it and return ACK;
      otherwise, reject it and return an ERR metagram;
    }
  }
}
```

Figure 3-11: *Pseudocode for the* ⌊File⌉ *application's* FileHandler.

```
class FileApplication extends FrameworkApplication {
  //  constructs a new instance of the File application,
  //  automatically finding and plugging in the handler
  public constructor FileApplication() { super("File"); }
  //  returns the single-user action for this application
  public ApplicationAction[] getBuddyActions(Identity id) {
    return new ApplicationAction[] { new SendFileAction(id) };
  }
}
```

Figure 3-12: *Pseudocode for the* ⌊File⌉ *application's framework interface.*

data representation. Although the framework's primitives are of prime importance, it is important to recognize the primitives eliminate a tremendous amount of code and complexity, and, in the process, greatly simplify application development.

3.4 Unimplemented Details

Some minor aspects of the framework design have not yet been implemented. Specifically, the following ideas are not currently reflected in the implementation:

1. The implementation does not use a CA at all. New nodes can join freely by contacting any node within the framework.

2. There is no automatic translation between application-level data and network-level (metagram) data representations. Application developers currently create and work with metagrams.

3. The peer channel descriptor mutators for encryption, authentication, reliability, and integrity-checking are not implemented. That is, one cannot currently change these channel options; all communications are encrypted, authenticated, reliable, and integrity-checked but not compressed or anonymized.

Chapter 4

Related Research

This framework builds upon a sizeable body of related work in several fields. The framework shares many goals with other peer-to-peer service offerings, and I compare and contrast their designs with the framework's design in Section 4.1. The framework's abstraction builds and improves upon many previous communication abstractions. I compare the framework's application primitives to other communication abstraction methods in Section 4.2.

4.1 Peer-to-Peer Services

Castro et al. have similar ideas about peer-to-peer application/service agglomeration [2] and suggest that all nodes participate in a universal overlay to tackle the difficult questions of "Who is running what and where?" and "How can I join?" However, each application runs in a separate overlay, which creates significant overhead for nodes running many applications. I have shown that there are many instances in which multiple applications can share a single overlay. I believe that a single overlay is the wrong abstraction for a peer-to-peer application, and the application-overlay interface should be as modular as possible. This independence is especially important

in times of rapid progress; applications should be written with flexibility in mind. New lookup technologies are likely to be developed, and inflexible applications will be pushed toward irrelevance. Also, applications built upon the universal ring will need their own security and communication schemes, whereas applications built upon the framework can reuse the framework's primitives. Finally, it is unclear what motivation nodes have to participate in this universal overlay for the vast majority of the time when they are not using its information. Nonetheless, their ideas for service discovery and code distribution seem valuable, and my work could be extended to incorporate solutions to these problems.

Stoica et al. propose *i*3 [18], a system that uses indirection to decouple the sender from the receiver. Much like the framework, *i*3 is an overlay network designed to enable distributed application development. Whereas the framework aims to simplify application development by eliminating security and networking complexity, *i*3 aims to provide support for non-point-to-point communication paradigms such as mobility, multicast, and anycast. A future version of the framework could incorporate *i*3's indirection technique to support these communication paradigms.

Castro et al. propose Scribe [29], a peer-to-peer, application-level multicast infrastructure. Scribe is built upon the Pastry peer-to-peer lookup protocol and focuses on enabling large numbers of multicast groups, large membership within a group, and many-to-many communication. Again, whereas the framework aims to simplify and improve point-to-point application development, Scribe support primitives only to make multicast scalable and reliable.

Freedman and Morris propose Tarzan [30], a peer-to-peer anonymizer that operates at the IP level. Tarzan and the framework's [WAnonymity] application share the fundamental idea of using Chaum's mix-nets within a peer-to-peer overlay to provide anonymity via multi-hop routing. Whereas [WAnonymity] is a proof-of-concept application and only attempts to provide weak anonymity, Tarzan has a security model with probabilistic anonymity guarantees and attempts to provide strong anonymity,

including cover traffic for known traffic analysis attacks. A future version of the framework could incorporate Tarzan's strong anonymity mechanisms to further support anonymity in its point-to-point communications.

4.2 Communication Abstraction

Metagrams are similar to the S-expressions of LISP, which were recently expanded by Rivest [14]. However, metagrams differ from S-expressions in that all metagram data is explicitly labeled. Metagrams are also similar to XML documents, and metagram exchange is similar to XML-RPC [19] and SUN RPC [20]. Metagrams are designed to represent any type of data, whereas XML was designed to mark up text-based documents. As such, XML inefficiently stores binary data encoded in base-64, resulting in binary XML documents that are one-third larger than necessary. Metagrams are inherently binary, require attributes to be nested subelements, and do not have any of the extra baggage that XML entails. Metagrams are also notably similar to AOL Instant Messenger's Type-Length-Value (TLV) data type [15]. AOL TLVs do not allow nesting and are thus quite limited as a representational format.

Peer channel descriptors are similar in essence to SSL [21] and Java's RMI [22]. SSL has design incompatibilities that make it an awkward choice for the framework's communication abstraction. In order for two parties to communicate securely, despite knowing nothing about one another, SSL requires a hierarchical certificate model for authentication. The framework has no possibility of out-of-band key exchange; joins would need to involve an authentication certificate in addition to the uniqueness certificate. Using these authentication certificates, identities could verify each other in a secure manner. However, this design gives the CA significantly more power, whereas in the framework's design, the CA merely limits the number of identities a single user can obtain.

Both RMI and Java's native streams for manipulating encryption, integrity, compres-

sion, and the like [23, 24, 25] are alternate solutions to the framework's communication abstraction. However, when I described the framework's properties in Section 1.1, I required technology independence. A solution involving RMI or Java's native stream manipulation would create the dependency that the implementation must be in Java. Many operating systems, including FreeBSD [26] and OpenBSD [27], still do not have native support for Java 2, and as such, a native implementation could not be created. Currently, the communication abstraction is free from such dependencies and is platform- and programming language-independent. This ensures that developers on any platform can implement the framework in any language.

Chapter 5

Conclusion

I explained my motivation in the first chapter: to create a peer-to-peer framework that enables superior applications to be developed more easily and more quickly. I analyze these claims in Section 5.1, and I discuss avenues for future work in Section 5.2, including extensions to the framework's fundamental capabilities. Finally, I conclude with a summary in Section 5.3.

5.1 Analysis of Framework Design

In Section 1.1, I delineated several design goals for the framework and claimed that a framework with these properties would result in superior applications. I now justify that the design presented in Chapter 2 satisfies the framework's properties, remains flexible and extensible to a variety of applications, and greatly simplifies application development.

1. *Security.* Abstraction of user-to-user communication allows the incorporation of already established point-to-point communication research. The point-to-point communication paradigm is well-understood, and as such, the framework's

point-to-point communication primitives support authentication, confidentiality, reliability, integrity checking, weak anonymity, and compression, each of which can be toggled on or off as desired for each block of data. Each application can use or not use these primitives, but the point-to-point communication of the framework's applications is secure by default.

2. *Decentralization.* The framework is completely decentralized, except for the process of certifying the uniqueness of a new node. An instance of the framework can skip this certification if it does need to locatable within the framework.

3. *Scalability.* The two lookup protocol applications, ⌊Ring⌉ and ⌊Blob⌉, used by the framework are both massively scalable. Bandwidth and storage constraints are almost non-existent due to the framework's decentralized, distributed nature.

4. *Technology Modularity and Independence.* Applications are not bound to lookup protocols, and lookups are not bound to applications. Abstracting away the underlying network allows for a diversity of lookup technologies. Preserving application independence from network technologies is especially critical for peer-to-peer applications, as applications will be able to migrate with rapidly developed new technologies and remain competitive. Similarly, no single lookup technology is perfect for all possible applications. Multiple technologies must be available to support the diversity of present and future peer-to-peer applications.

   By relying only on an abstraction of the underlying network, framework applications remain adaptable to even the most fundamental changes to the framework. This abstraction also improves efficiency, as applications do not need their own overlay. Despite the "chattiness" of peer-to-peer applications, the number of framework applications that can run simultaneously on the same machine is not bounded by their peer-to-peer nature.

   The framework has other key flexibilities. Identities are not bound to locations like in other systems, such as Chord [11], allowing users to be mobile within

their underlying network connections. Also, the design of the framework is not bound to programming languages or operating systems or even the IP layer, and this technology independence theoretically allows instances of the framework to run on any type of networked device, including suitably powerful cell phones and PDAs.

5. *Technology Transparency.* All aspects of the framework are open, and hence, understandable and criticizeable. The benefits of openness extend to the application domain as well. Applications can reuse one another's functionality, as was evidenced repeatedly in the implemented applications in Section 3.1. One example of this potential is application composition, which is possible because metagram exchange is a powerful and general substrate for constructing distributed applications.

   For instance, application A can process a metagram, modify it, and then enter it back into the metagram dispatcher formatted such that it will then be handled by application B. In this manner, an application can be composed with other applications, much like the command line utilities of UNIX.[1] Such application composition is quite practical, given that all the framework's applications are manipulating data streams. Simple applications whose sole purpose is performing data translations, such as data compression or real-time video reformatting, would be quite useful both as independent applications or within larger composite applications.

   Applications need not be composite in order to be useful to one another. If application A exports some of its basic functionality as services, then application B can incorporate this functionality and need not re-implement it elsewhere. In this sense, the *openness* applications that export services create new framework primitives upon which future applications can build.

6. *Popularity.* The framework's primitives greatly simplify peer-to-peer applica-

[1] An astute observer might quip that the "everything is a metagram" philosophy is quite similar to the "everything is a file" philosophy in UNIX.

tion development. This simplification allows a significantly larger range of developers to create peer-to-peer applications, while simultaneously allowing them to create the applications faster and acclimate themselves to the framework more quickly. These developers will create new applications, which will in turn draw more developers to the framework.

In a fashion somewhat reminiscent of the IP layer's clean separation of network and application, the framework's abstraction decouples applications from the underlying network. This allows the framework to accommodate a wide variety of applications and provide core peer-to-peer functionality that applications can reuse, while placing little constraint on the types of applications that can be built. The result is a framework that produces superior applications (with more features, fewer bugs, and fewer insecurities) faster and easier (using shorter and simpler code) than building directly upon the IP layer. These results are corroborated by the simple and concise code of the implemented applications.

5.2 Future Work

While the framework offers substantial benefits for peer-to-peer application development, its design can be expanded to allow a wider range of applications to be constructed more easily. A future version of the framework could encompass the following capabilities:

1. *Framework-level data storage.* A workable solution to this problem would improve the framework by significantly simplifying the construction of applications that require distributed data storage. Unfortunately, framework-level data storage is a complex problem that is compounded further by the multiple ways in which the framework can reasonably store data. Each lookup protocol could

easily be extended to store and look up data items as well as identities; however, overlays would store data in drastically different ways.

For instance, in a comparison of ⌊Ring⌉ and ⌊Blob⌉, the Chord-like and Freenet-like lookup protocols respectively, one major difference would be data persistence. Data stored in ⌊Blob⌉ would not be guaranteed to persist, but data stored in ⌊Ring⌉ would be. Similarly, there would be availability differences. If the desired data existed in ⌊Ring⌉, it would always be found, whereas in ⌊Blob⌉, the data may not be found even if it did exist. Similarly, there would also be latency differences. ⌊Ring⌉ is structured, and as such, the latency on a data lookup would be tightly bounded. ⌊Blob⌉ is unstructured, and the latency on its data lookups would frequently be as long as several minutes. Also, there would be anonymity differences. A node that held a data item in ⌊Ring⌉ could log each access to that data item, while ⌊Blob⌉'s algorithms are set up such that logging requests would be nearly impossible. Finally, there would be robustness differences, although both would attempt to provide robustness through redundancy. ⌊Ring⌉ would replicate each data item across k nodes to ensure that all data remains available with high probability. ⌊Blob⌉, on the other hand, would replicate data only when that data item is successfully requested.

Unfortunately, it is unclear if either of these potential solutions is workable. I have two main concerns with ⌊Ring⌉ data storage: (1) under highly volatile presence fluctuation, a tremendous amount of data might need to be transferred; and (2) since the data persists, storage needs are monotonically non-decreasing over time. It is also unclear if ⌊Blob⌉ is actually viable for many types of application data, because data is not guaranteed to be found and lookup latencies could be intolerably long.

2. *Support for multiple communication paradigms.* The framework only provides an abstraction for the point-to-point communication paradigm, where one user wants to communicate with exactly one other user. Other paradigms, known as *multicast* and *anycast*, involve one-to-many or many-to-many communication

paradigms.

In a similar manner to how the framework simplifies point-to-point application construction, framework-level abstractions of these other communication paradigms would make entire classes of distributed applications significantly easier to develop.

3. *Stronger anonymity.* Despite a wealth of research literature on anonymity, the anonymization methods implemented in the framework are not perfect. The strength of the framework's anonymity could be improved by incorporation of ideas from Tarzan [30] and Chaum's mix-nets [17], for instance.

4. *Code release.* I would eventually like to release my implementation to the world. Before I will do this, I need to be more certain that the implementation works as well as it should in theory. The networking code works well in experiments involving a handful of nodes, but my testing to date has not involved larger networks. Further stress testing and performance analysis are necessary.

5.3 Summary

This thesis has presented an application framework that greatly improves the peer-to-peer application development process. The framework abstracts networking and security complexity away from developers, who only need application-specific knowledge to build robust, scalable peer-to-peer applications. The superiority of the development process is evidenced by the example applications' implementations. Application source code is shorter and simpler, resulting in easier and faster application development with fewer possibilities for bugs and insecurities.

The framework also provides many qualitative contributions. Its ease of extensibility and abstraction of complexity opens the peer-to-peer community to a wider range of developers (and thus ideas) while ensuring that these applications will be secure,

robust, and follow many "best practices" of software engineering. The framework's openness encourages sharing of ideas, which in turn promotes learning, progress, and better applications for the end user.

This is, of course, the whole point of the framework: to create better applications that have more value for the end user. Framework applications create new and better methods of online interaction, which will be critical to human communication in the future. Well-connected users will be able to harness the power and connectivity of their computers to manage mundane detail, improve interpersonal communications, and, as a result, make their lives easier and better.

Appendix A

Point-to-Point Protocol

Suppose an identity named Alice wants to send a metagram to an identity named Bob. Alice and Bob know each other's identity, and Alice has a method of getting bits to Bob, but she has never communicated with Bob. Can they still communicate securely? The answer is yes, since messages from that appear to come from one's identity must have been created by that identity, as shown in Section 2.2.1. Alice and Bob use this property to accomplish secure communication through the framework's point-to-point protocol, which is similar to the protocol in RFC 2617 [4]. This appendix provides the detail about the three-part protocol that allows secure communication from this setup.

A.1 Stage 1 – Initialization

Alice and Bob begin a secure connection by exchanging 64 bits of random data and forming a 128-bit initialization vector for the first part of the protocol. This step is illustrated in Table A-1. This initialization vector will later serve to seed the encryption process with random data.

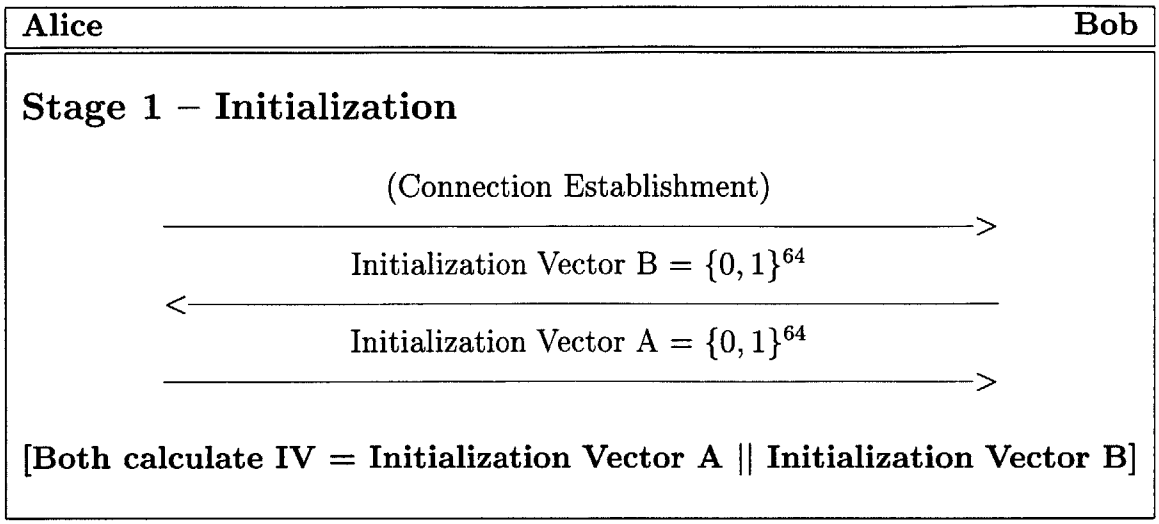

Figure A-1: *Stage 1 of the point-to-point protocol: Initialization.*

A.2 Stage 2 – Ticket Exchange

Next, Alice and Bob generate and exchange cryptographic *tickets* in a somewhat complicated process, which is shown in Figure A-2. Alice and Bob start by performing Diffie-Hellman key exchange and use the exchanged data to calculate two unidirectional session keys: $K_{\rightarrow}$ from Alice to Bob and $K_{\leftarrow}$ from Bob to Alice. These session keys will serve to encrypt future data sent to one another.

Next, Alice and Bob authenticate themselves by creating DSA signatures on the previously exchanged Diffie-Hellman data. Since only Alice and Bob can create their respective signatures, each is cryptographically assured of the other's identity. At this point, Alice and Bob could secure communicate metagrams to one another, but first, Alice and Bob create tickets that represent all the necessary data to re-establish this secure connection. The ticket Alice creates for Bob is encrypted with Alice's master AES key and then sent to Bob. Bob also encrypts his ticket and sends his ticket to Alice. The ticket format is described in Table A.1.

Instead of going through this entire ticket generation and exchange process, the next time Alice wants to communicate something to Bob, she can simply present to Bob

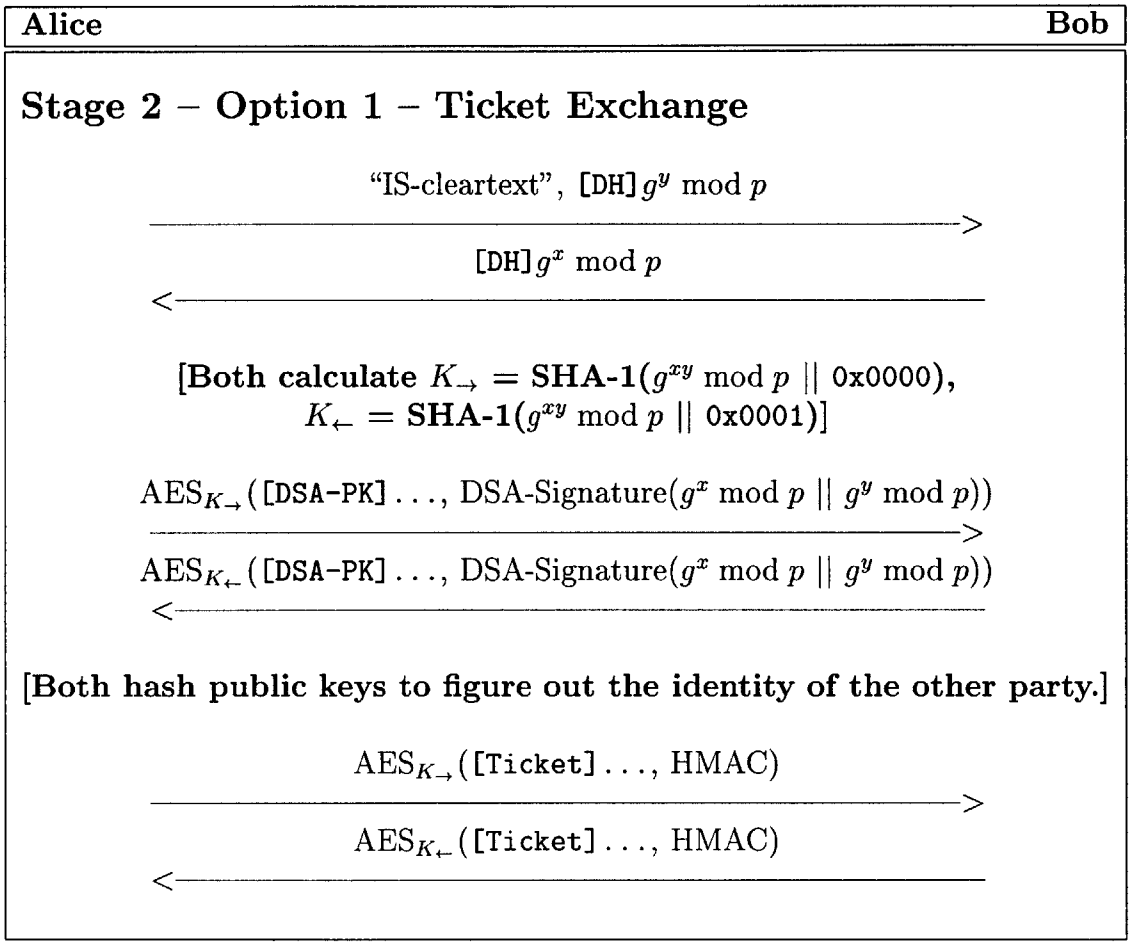

Figure A-2: *Stage 2, Option 1 of the point-to-point protocol: Ticket Exchange.*

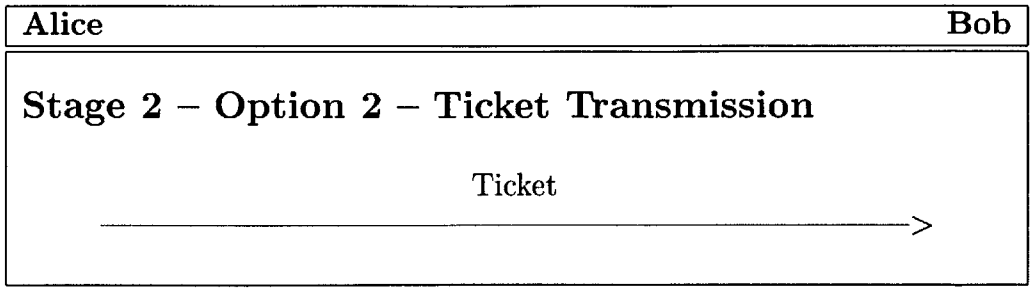

Figure A-3: *Stage 2, Option 2 of the point-to-point protocol: Ticket Transmission.*

the ticket he previously gave her. This is another option for the second stage of the protocol and is illustrated in Figure A-3; they can immediately resume the previously established secure connection instead of going through the computationally-intense cryptographic handshake listed in Figure A-2. This is possible because Alice has remembered the session keys, and Bob is able to decrypt the ticket and extract the session keys that will decrypt any content that immediately follows. Note that since the ticket is encrypted with the Bob's master AES key, only Bob can read it, and it is fine to transmit it to Bob in the clear.

A.3 Stage 3 – Metagram Exchange

At this point, there is an open socket between parties who share secret session keys. Both parties are free to send metagrams via this "encrypted socket" as shown in Figure A-4. Metagrams are sent as ⟨`CallTag, Metagram, HMAC`⟩ tuples, and many such tuples can be transmitted in both directions. Tuples whose HMACs[1] fail are silently

[1]The HMAC is a symmetric key message authentication code (MAC) that is computed by hashing the shared secret key, initialization vector, and the message (in our case `CallTag` followed by `Metagram`). The resultant hash and the secret key are hashed again to yield the HMAC, which guarantees the integrity of the message.

Field	*Identity*	$K_{\rightarrow}$ *Session Key*	$K_{\leftarrow}$	*Stop Time*	*Start Time*	*HMAC*
# of Bits	128	256	256	64	64	160

Table A.1: *Point-to-point protocol ticket format.*

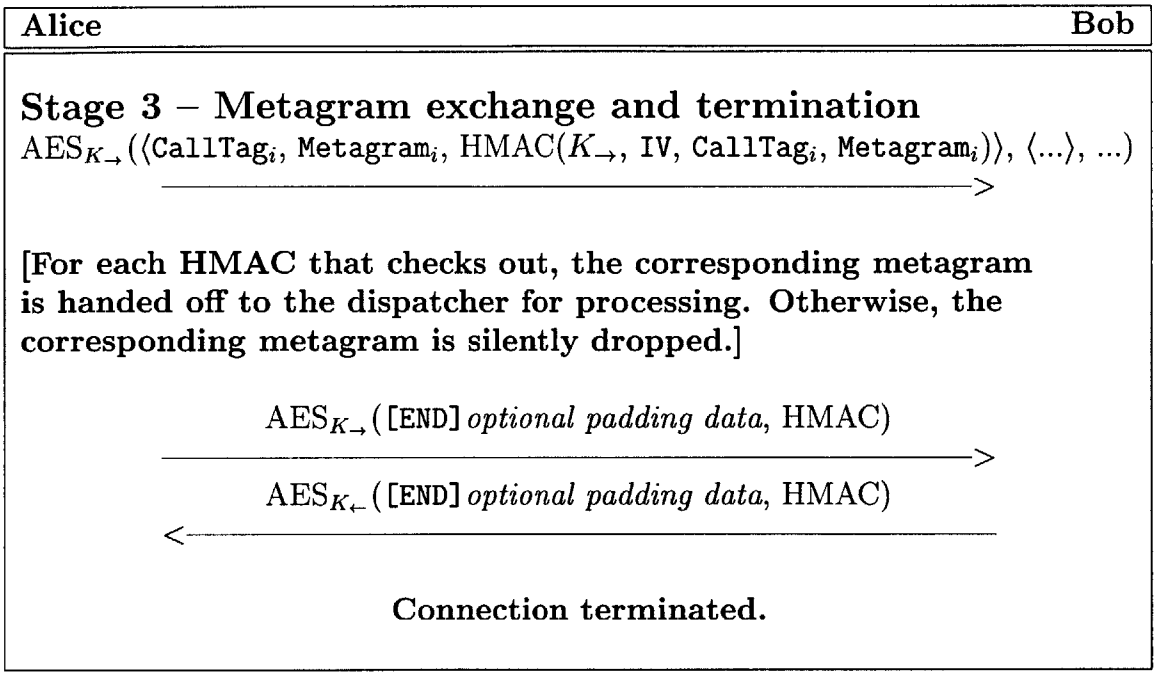

Figure A-4: *Stage 3 of the point-to-point protocol: Metagram Exchange.*

dropped. Otherwise, the metagram is bundled with its context, namely the sender's identity, and handed off to the metagram dispatcher and eventually to an application-level module for content processing. The application-level module eventually returns with a response metagram. The response metagram is forwarded back along the encrypted socket with its `CallTag` set as the request's `CallTag` but with the most significant bit flipped to indicate a return instead of an initial call.

A.4 Optimization Discussion

It is important to note that the ticketing optimization is just to make subsequent metagram exchanges as light as possible. Alice and Bob must do at least one computationally intensive cryptographic handshake to set up a secure connection. The ticketing optimization allows the parties to reuse the secure connection, greatly reducing the computational overhead involved in subsequent connections.

Appendix B

Peer Channel Descriptor API

A peer channel descriptor is the abstraction of how applications communicate with one another within the framework. It represents everything about a communication channel to another peer, including the peer on the other side as an identity, the location of the identity,[1] and parameters that affect how the data is transmitted across the network and how the peer is located within the framework.

One creates an instance of a peer channel descriptor with one of several constructors. Upon creation, data can immediately be sent to the other peer in the default manner via the `send` method. The `send` method call executes the lookup process if the peer's location is not already known. Alternatively, one can specify a non-standard lookup process before establishing a peer's location via a `send` method call.

Peer channel descriptors also maintain communication options such as toggling encryption, authentication, or compression that specify how future data should be sent. These options are orthogonal and can be independently turned on or off as desired.

[1]For all intents and purposes, locations within the framework are IP addresses. However, it is important to note that the location of a framework identity is not limited to just IP. If the framework's underlying platform had a way to represent and communicate with locations in other networks besides the Internet, identities could easily communicate across these networks within the original design of the framework.

B.1 API Specification

The Peer Channel Descriptor API consists of constructors to create peer channel descriptor objects, a method to see if a peer is currently online, methods to manage the process of peer location, a method to send data to the peer, and methods to manage how data is sent to the peer.

B.1.1 Constructors

All peer channel descriptor objects are created through the constructors listed in Table B.1 or through constructors defined in application-specific peer channel descriptor subclasses. Constructors' parameters uniquely identify peers within the framework, and different constructors use different procedures to create full peer channel descriptor objects from these parameters.

Most peer channel descriptors will be instantiated via the first constructor, which creates an instance using only a framework identity. Once an instance created with this constructor needs to send data, the peer channel descriptor will automatically use the framework's lookup protocols to locate the identity before sending. The details of how the lookup is performed and can be configured are explained in Section B.1.3.

The parameters of the second and third constructors are in the form of a peer's location as an IP address and a TCP port or a DNS name and a TCP port. These constructors are used when peers are specified directly by their network addresses, such as in lookup protocol applications, as well as when a node does its initial framework discovery. Note that the peer's identity is not discovered until the first communication.

The fourth constructor is similar to the first, as a public key uniquely specifies an identity. The public key is hashed to get the peer's identity, and then the peer channel

`PeerChannelDescriptor(identity)` Defines a peer channel descriptor (with the default settings) for the given identity.
`PeerChannelDescriptor(ip-address, port)` Defines a peer channel descriptor (with the default settings) for the given IP address and TCP port.
`PeerChannelDescriptor(dns-name, port)` Defines a peer channel descriptor (with the default settings) for the given DNS name and TCP port.
`PeerChannelDescriptor(public-key)` Defines a peer channel descriptor (with the default settings) for the given public key.

Table B.1: *Peer channel descriptor constructors.*

`boolean`	`isAlive()` Returns true iff the identity corresponding to this peer channel descriptor is currently locatable.

Table B.2: *Peer channel descriptor presence establishment.*

descriptor is created using the identity.

B.1.2 Presence

Applications frequently need to establish the presence of a peer, i.e., whether the peer is currently reachable for communication. Peer channel descriptors have such a method, listed in Table B.2.

B.1.3 Lookups

Frequently, the location of an identity within the framework needs to be ascertained. Peer channel descriptors discover the location using a process that is straightforward but customizable, and as such many parameters are involved.

Under normal circumstances, a peer channel descriptor uses the framework's default lookup protocols in order to find identities. However, this behavior is just the de-

`Lookups`	`getLookups()` Returns the lookup list
`void`	`clearLookups()` Clears the lookup list.
`void`	`addLookup(lookup)` Adds lookup to the lookup list.
`void`	`removeLookup(lookup)` Removes lookup from the lookup list.

Table B.3: *Peer channel descriptor lookup list operations.*

`boolean`	`isLookupAllMode()` Returns true if the peer channel descriptor will try to lookup the given node ID by all possible means. The default value is false.
`void`	`setLookupAllMode(boolean)` Changes the flag for trying all possible lookups.
`boolean`	`isLookupDefaultMode()` Returns false if the PCD will only use the lookups specified in its lookup list and will not fall back on default lookups. The default value is true.
`void`	`setLookupDefaultMode(boolean)` Changes the flag for trying the default lookups (after the specified lookups fail).

Table B.4: *Peer channel descriptor lookup modes.*

fault. Applications may change this behavior by customizing the lookup process of a peer channel descriptor instance. Peer channel descriptors have a *lookup list*, which manages an ordered list of lookup protocols that will be used in order to locate an identity. Next, depending on the peer channel descriptor's *lookup-defaults* mode, the framework's default lookups may be tried. Finally, depending on the *lookup-all* mode, the framework's remaining lookups may be tried. Methods for specifying the lookup list are specified in Table B.3, methods for altering the lookup modes are specified in Table B.4, and pseudocode for this identity lookup process is delineated in Figure B-1.

B.1.4 Communication parameters

Peer channel descriptors also specify the manner in which data is sent. Data can be sent with any combination of the following orthogonal parameters set: encryption, authentication, integrity-checking, compression, reliability, and anonymity. The methods for accessing and toggling these parameters as well as their default values are listed in Table B.5.

B.1.5 Data Transfer

The final duty of a peer channel descriptor is the actual transmission of application data to the peer. The `send` method, listed in Table B.6, executes the transfer of `data` to the peer according to its communication parameters using the point-to-point protocol and returns the peer's reply: `response`.

B.2 Pseudocode

The pseudocode for this entire API is rather trivial except for the method that finds the location for an identity. This pseudocode for this method is listed in Figure B-1.

`boolean`	`isEncrypted()` Returns true if this PCD encrypts all data sent through it. The default value is true.
`void`	`setEncrypted(boolean)` Toggles the encryption flag on this channel.
`boolean`	`isAuthenticated()` Returns true if this PCD authenticates all data sent through it. The default value is true.
`void`	`setAuthenticated(boolean)` Toggles the authentication flag on this channel.
`boolean`	`isIntegrityChecked()` Returns true if this PCD integrity-checks all data sent through it. The default value is true.
`void`	`setIntegrityChecked(boolean)` Toggles the integrity-checking flag on this channel.
`boolean`	`isCompressed()` Returns true if this PCD compresses all data sent through it. The default value is false.
`void`	`setCompressed(boolean)` Toggles the compression flag on this channel.
`boolean`	`isReliable()` Returns true if this PCD reliably (TCP) sends data through it, false if unreliably (UDP). The default value is true.
`void`	`setReliable(boolean)` Toggles the reliability flag on this channel.
`boolean`	`isAnonymized()` Returns true if this PCD anonymizes all data sent through it. The default value is false.
`void`	`setAnonymized(boolean)` Toggles the anonymity flag on this channel.

Table B.5: *Peer channel descriptor communication parameters.*

`response`	`send(data)` Sends data to the other peer using the point-to-point protocol (specified in Appendix A) in the manner specified by the options of the peer channel descriptor.

Table B.6: *Peer channel descriptor data transfer.*

```
private Location getLocation() {
    if location is already known, return location;
    triedLookups = { };
    //  first try lookups in specified lookup list
    lookupList = this.getLookupList();
    for(int i=0; i<lookupList.length; i++) {
      lookup = lookupList[i];
      if triedLookups contains lookup, continue;
      location = Framework.executeLookup(identity, lookup);
      if location is now known, return location;
      add lookup to triedLookups;
    }
    //  next see if the default lookups should be tried
    if isLookupDefaultMode() {
      defaultLookups = Framework.getDefaultLookups();
      for(int i=0; i<defaultLookups.length; i++) {
        lookup = defaultLookups[i];
        if triedLookups contains lookup, continue;
        location = Framework.executeLookup(identity, lookup);
        if location is now known, return location;
        add lookup to triedLookups;
      }
    }
    //  next see if all lookups should be tried
    if isLookupAllMode() {
      allLookups = Framework.getAllLookups();
      for(int i=0; i<allLookups.length; i++) {
        lookup = allLookups[i];
        if triedLookups contains lookup, continue;
        location = Framework.executeLookup(identity, lookup);
        if location is now known, return location;
        add lookup to triedLookups;
      }
    }
    //  we have exhausted all lookups without success
    return null;
}
```

Figure B-1: *Pseudocode for peer channel descriptor identity location.*

Appendix C

IMHandler Code

I have included the `IMHandler` code so that one can see actual code (and not pseudocode) that represents the entire functionality of the [IM] application.

Note the brevity and simplicity of the code.

```
package is.apps.im;

import is.*;
import is.apps.*;
/** IMHandler is the implementation of the ApplicationHandler interface
 * for processing [IM] metagrams for the (very simple) IM
 * application. */
public class IMHandler implements ApplicationHandler {
  /** This HashMap is the list of currently active conversations.  The
   * keys are the node IDs.  The values are the GUI panels holding the
   * conversations. */
  private java.util.HashMap hmConversations;
  /** Defines a new IMHandler with no current conversations. */
  public IMHandler() { hmConversations = new java.util.HashMap(); }
  /** Signifies that this handler handles [IM] metagrams. */
```

```
  public String getName() { return "IM"; }
  /** This method handles a context metagram whose content metagram is
   * an [IM]. If the IM is received successfully, this method returns
   * an [ACK]. If there is a problem, this method returns an
   * [ERR]. */                                                          20
  public Metagram handle(ContextMetagram cm) {
    if(cm == null) return is.net.Network.mgERR;
    NodeID nodeID = cm.getNodeID(); // get sender
    Metagram mgIM = cm.getMetagram(); // get message
    // See if we already have a conversation going with this node.
    IMPanel imp = (IMPanel)hmConversations.get(nodeID);
    if(imp == null) imp = (IMPanel)initiate(nodeID); // new conversation
    imp.receive(mgIM);
    return is.net.Network.mgACK;
  }                                                                     30
  /** Initiate an IM conversation to the given node. */
  public is.gui.ISPanel initiate(NodeID nodeID) {
    IMPanel imp = new IMPanel(nodeID);
    hmConversations.put(nodeID, imp);
    imp.showDialog();
    return imp;
  }
  /** Close an IM conversation to the given node. */
  public void close(NodeID nodeID) {
    hmConversations.remove(ISUtils.toString(nodeID.getBytes()));        40
  }
}
```

Bibliography

1. Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. *Security for structured peer-to-peer overlay networks.* Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, MA, December 2002.

2. M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron. *One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks.* SIGOPS European Workshop, France, September, 2002.

3. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. *Freenet: A Distributed Anonymous Information Storage and Retrieval System.* Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, New York, 2001.

4. W. Diffie, P.C. van Oorschot, and M.J. Wiener. *Authentication and Authenticated Key Exchanges.* Designs, Codes, and Cryptography, v.2, 1992, 107-125. (RFC 2617)

5. John R. Douceur. *The Sybil Attack.* Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts, March 2002.

6. S.P. Miller, B. C. Neuman, J. I. Schiller, and J.H. Saltzer. Section E.2.1: *Kerberos Authentication and Authorization System.* Project Athena Technical Plan, MIT Project Athena, Cambridge, MA, October 1988. (Version 4)

7. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. *A Scalable Content-Addressable Network.* Proceedings of ACM SIGCOMM '01, San Diego, California, August 2001.

8. R. Rivest, A. Shamir, and D. Wagner. *Time-Lock Puzzles and Timed-Release Crypto.* MIT/LCS/TR-684. February, 1996.

9. Antony Rowstron and Peter Druschel. *Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems.* Proceedings of ACM SOSP '01, Banff, Canada, October 2001.

10. Emil Sit and Robert Morris. *Security Considerations for Peer-to-Peer Distributed Hash Tables.* Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts, March 2002.

11. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts, March 2002.

12. Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. *Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing.* Technical Report UCB//CSD-01-1141, UC Berkeley, April 2001.

13. S.M. Bellovin. *Security Problems in the TCP/IP Protocol Suite.* Computer Communication Review, Vol. 19, No. 2, pp. 32-48, April 1989.

14. R. Rivest. *SEXP—(S-expressions).*
`http://theory.lcs.mit.edu/~rivest/sexp.html`.

15. ——. *Sourceforge AOL Instant Messenger Specification.*
`http://aimdoc.sourceforge.net/`
`http://www.aim.aol.com/javadev/terminology.html`

16. Petar Maymounkov and David Mazieres. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.* Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts, March 2002.

17. David L. Chaum. *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms.* Communications of the ACM, 24(2):84-88, February 1981.

18. Ion Stoica, Dan Adkins, Sylvia Ratnasamy, Scott Shenker, Sonesh Surana, and Shelley Zhuang. *Internet Indirection Infrastructure.* Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, Massachusetts, March 2002.

19. XML-RPC. `http://www.xmlrpc.com/`

20. ONC (SUN) RPC. RFC 1831: `http://www.ietf.org/rfc/rfc1831.txt`

21. SSL. `http://www.openssl.org/`

22. Java RMI. `http://java.sun.com/products/jdk/rmi/`

23. Java Security. `http://java.sun.com/j2se/1.4.1/docs/api/java/security/package-summary.html`

24. Java Crypto. `http://java.sun.com/j2se/1.4.1/docs/api/javax/crypto/package-summary.html`

25. Java Compression. `http://java.sun.com/j2se/1.4.1/docs/api/java/util/zip/package-summary.html`

26. FreeBSD. `http://www.freebsd.org/`

27. OpenBSD. `http://www.openbsd.org/`

28. OpenAFS. `http://www.openafs.org/`

29. M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron. *SCRIBE: A large-scale and decentralised application-level multicast infrastructure.* IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications). 2002.
`http://research.microsoft.com/~antr/SCRIBE/`

30. Michael J. Freedman and Robert Morris. *Tarzan: A Peer-to-Peer Anonymizing Network Layer.* Proceedings of the ACM Conference on Computer and Communications Security (CCS 9). Washington, D.C. Nov. 2002.
`http://www.pdos.lcs.mit.edu/tarzan/overview.html`

31. Stuart Staniford, Vern Paxson, and Nicholas Weaver. *How to 0wn the Internet in Your Spare Time.* Proceeding of the 11th USENIX Security Symposium. 2002.