Yale University
Department of Computer Science

P.O. Box 208205
New Haven, CT 06520–8285

Typed Computational Email for Serverless Distributed Applications

Robert Fischer[1] *Harvard University*
Hong Jiang[2] *Yale University*
Michael Fischer[3] *Yale University*

YALEU/DCS/TR-1291
May 19, 2004

[1]Email: rfischer@rics.bwh.harvard.edu
[2]Email: hong.jiang@yale.edu
[3]Email: fischer-michael@cs.yale.edu

Typed Computational Email for Serverless Distributed Applications

Robert Fischer[*] *Harvard University*
Hong Jiang[†] *Yale University*
Michael Fischer[‡] *Yale University*

Abstract

Peer agents are executable object-oriented email components exchanged between users as a basis of communication for distributed applications. Peer agents, which we implement, are strongly typed and are dispatched to user-installable trusted handlers based on their type. The type mechanism provides the flexible trust management needed to implement secure distributed distributed applications over store-and-forward networks. Peer agents address a number of contemporary problems in distributed systems. They allow users with low-grade or intermittent Internet access to provide and consume interactive services that typically require a web server hosted by a trusted third party. By eliminating the need for the trusted third party, privacy is enhanced. Peer agents allow an initiator to adopt new standards without requiring prior agreement with others involved in the communication. For all of these reasons, peer agents offer an attractive alternative to conventional client/server architectures for distributed applications.

1 Introduction

1.1 A Fable of our Times

Alice, who possesses a computer with a dial-up Internet connection, is planning a party in her home and will be inviting dozens of guests. She wishes to send out invitations electronically and to compile a list of responses. To automate this task, she uses a popular web service, which we will call *myparty.com*.[1] She logs onto *myparty.com* and enters her guest list. The web site sends email notifications to her guests.

Bob, one of Alice's guests, receives the notification. It tells him that Alice has invited him to a party, but in order to view and respond to the invitation, he must click on a URL that takes him to the *myparty.com* web site. There, he will learn the details of the party and be able to indicate whether or not he intends to come. Not wishing to be seen browsing the web at work, Bob puts the email aside without viewing the invitation.

A week later, during an airplane flight, Bob is using his PDA to sort through his deferred email. He comes across the notification from *myparty.com* and remembers that Alice invited him to her party. He would like to go but is still not sure when the party will take place; without Internet access, he is unable to make further progress on this issue.

Late the next day, Bob finally finds time to get on-line and respond to Alice's invitation. He decides he will go and enters that information into *myparty.com*, even though the RSVP date has already passed. Later, Alice logs onto *myparty.com* to view the response list for her party. She is pleased to see that Bob, a busy man, has finally decided to come, although she will have to change her plans to accommodate his late RSVP.

Unbeknownst to Alice, one of Alice's guests is a friend of a suspected enemy of the state. Eve, working for the government's wire-tapping agency, subpoenas *myparty.com* for all information regarding Alice's

[*]Email: rfischer@rics.bwh.harvard.edu
[†]Email: hong.jiang@yale.edu
[‡]Email: fischer-michael@cs.yale.edu
[1]No relationship with the domain by the same name that is registered by New Net Companies, Inc. is intended.

party. Shortly after the party, Bob is questioned by men in dark glasses. Eve has found the extensive social networks tracked by *myparty.com* to be quite useful for her work. Meanwhile, Alice and her guests begin to receive increased spam in their inboxes, for *myparty.com* has chosen to sell their email addresses.

1.2 Shortcomings of the Web-Based Approach

Our example of Section 1.1 illustrates two shortcomings of the hypothetical *myparty.com*:

- The privacy of Alice's guest list relies on trust she places in a third party, *myparty.com*. The fable shows two reasons why *myparty.com* might violate that trust—because of legal requirements and for commercial gain.
- Bob is required to access a web site in order to respond to Alice, even though the asynchronous store-and-forward communication provided by ordinary email would seem to be sufficient for returning an RSVP. This was inconvenient for Bob, who experienced intermittent Internet access due to company policy and travel, and delayed his response to Alice.

So why does Alice use *myparty.com*? First, it provides her with a convenient web form and database for compiling her guest list. Second, it provides an interactive experience to Bob. Most important, it automatically organizes and tabulates RSVP responses for Alice—a big time saver. Although she is concerned about the privacy implications of giving her personal data to a third-party web site, she does not have the means to set up a web server on her own computer. She chooses to compromise privacy for convenience.

1.3 Peer Agents

Peer agents offer an alternative approach to providing many of the same kinds of applications that web servers provide. In particular, they can be used to provide the kind of party invitation service desired by Alice in the fable.

A *peer agent* is an executable object-oriented email component exchanged between users as a basis of communication for distributed applications. Peer agents are strongly typed and are dispatched to user-installable trusted handlers based on their type. A *peer agent application* is a distributed application implemented through a combination of locally installed software and peer agent handlers.

The action of a handler on a peer agent depends entirely on the handler. A handler dictates its own security policy in executing a peer agent. In our Java implementation, it has the entire Java security and cryptography systems at its disposal, allowing it to authenticate, sandbox, encrypt and decrypt peer agents as needed, depending on the application. In sandboxing, it can provide peer agents with any set of primitives it likes, depending on the level of trust deemed appropriate for a given peer agent.

Although peer agent types and handlers are application-dependent, all peer agent systems are able to process peer agents of type *untrusted*, allowing receipt of peer agents without prior arrangement with the sender. The handler for *untrusted* type peer agents runs them in a restrictive sandbox: local and network I/O are prohibited, although the peer agent *is* allowed to interact with the user through a GUI and to construct and send a reply peer agent. Thus, peer agents provide a flexible framework that allows custom distributed applications to be run in an email environment. Security and trust policies are application-specific.

1.4 Party Planning with Peer Agents

We now examine how Alice could have planned her party using peer agents and a locally installed party planning application, *MyParty*. She enters her guest list into the application, which runs on her machine. *MyParty* then emails a peer agent to each of her guests. When her guest Bob receives a peer agent of

type *untrusted* from Alice, Bob's system dispatches it to the *untrusted* handler. The peer agent displays an animated invitation and asks for Bob's reply, which he enters. An *RSVP*-type peer agent is then constructed and sent back to Alice in reply.

When Alice receives the *RSVP*-type peer agent, it is dispatched to the *RSVP* handler, which was installed locally by the *MyParty* application. The *RSVP* handler first checks that the peer agent is an authentic reply from Bob to her invitation. If so, the *RSVP* peer agent is allowed to record a response in Bob's place—and only in Bob's place—of Alice's response list. In this way, guest responses are automatically tabulated, no trusted third party is invoked, and all transactions take place over store-and-forward email. Later, Alice can run the *MyParty* application to view her acceptance list.

Note that Alice, the *initiating party* of the communication, needs to install software and handlers on her system to use *MyParty*. In contrast, Bob, a *receiving party*, does not need to install anything special beyond the basic peer agent system. This is commonly the case for peer agent applications, that only the initiating party needs to install software to use an application, while receiving parties can participate using generic code. This property of peer agent applications is similar to the situation with many web applications such as e-commerce sites, where all application-specific code resides on the web server and users do not need to install application-specific client software in order to use them.

1.5 Related Work

Peer agents are an outgrowth of computational email, which was mentioned but not implemented at least as early as 1976 [1]. Early computational email systems suffered serious flaws, including lack of generality, lack of conformity to standard store-and-forward protocols, lack of security, and lack of cross-platform portability [2]. These problems were first solved with the ATOMICMAIL system [2] in 1992. Safe-Tcl [3, 14, 19] added an additional level of extensibility and GUI functionality. Sandboxing was just beginning to become common at that time: much of the work on ATOMICMAIL and Safe-Tcl went into building safe sandboxed languages. More recently, SHOCK [18] uses computational email for specialized purposes.

Although they have been around for over a decade, none of the above approaches for general computational email has seen widespread use. Past systems allow for only one type of computational email, and correspondingly one security policy, per installation. This limited their ability to support many useful applications.

Consider the party planning example again, in which the *RSVP*-type peer agent writes to Alice's response list. One would not normally include a primitive to write to this file in a "general" sandbox since that would allow an arbitrary user to impersonate Alice's guests. However, it is exactly this ability for an *RSVP*-type peer agent to modify Alice's acceptance list that makes the *MyParty* application work in a natural and useful manner.

The use of types and handlers for peer agents is similar to the use of types for non-computational email attachments. MIME types [12] are now in widespread use as a way to build extensible (non-computational) email clients. Every email attachment is associated with a MIME type. The email client maintains an association between MIME types and local applications. When handling an attachment, the email client passes the attachment to the application that matches the attachment's MIME type. Simple and straightforward, MIME types have had a profound effect on the content and utility of email.

Peer agents combine the computational aspects of computational email with the typing idea of MIME types to create a flexible basis for distributed applications over store-and-forward networks. The addition of types to computational email greatly enhances the utility of the paradigm.

```
public interface Handler
{
    // The peer agent type with which this handler associates
    public String getPeerAgentType();
    // The expected class of peerAgentObject (argument to handlePeerAgent())
    public Class getDefiningClass();
    // Called to handle a peer agent
    public void handlePeerAgent(Object peerAgentObject);
}
```

Figure 1: The *Handler* interface, implemented to create a locally installed peer agent handler.

1.6 Organization of This Paper

The rest of this paper is organized as follows. Section 2 describes the technical details of peer agents and agent applications. Section 3 briefly mentions some promising uses for peer agents and future research directions. We conclude in Section 4.

2 Peer Agent Framework

We have implemented a prototype peer agent system [9], written in Java [5]. Java was chosen for its flexible sandboxing model, its easy object serialization,[2] its cross-platform compatibility, and its extensive class libraries including GUI and cryptographic components.

Peer agents are normally transported as email attachments. In some cases, peer agents might be distributed in other ways—downloaded from a web site or even mailed on a CD-ROM.

A peer agent is a data object consisting of three parts:

1. The peer agent *type*, represented by a string. This type is used to dispatch the peer agent to its appropriate *handler*.
2. A set of resources, which in the Java implementation are class files (compiled Java bytecode).
3. The serialized *peer agent object*.

Similarly, a handler is an object consisting of three parts. It is created by implementing the *Handler* interface shown in Figure 1. The parts are as follows:

1. The handler's *type*. It is represented by a string. The handler is associated with this type, causing the peer agent framework to dispatch peer agents of the same type to it.
2. The *defining class* for the handler (and for the handler's type). Peer agents objects dispatched to this handler are expected to be a subclass of the defining class. It is customary to name the defining class after the handler's type.
3. The *handlePeerAgent()* method. It is called to dispatch a peer agent to a handler.

When a peer agent arrives on a system, it is processed as follows:

1. The email client, web browser or other system program, in accordance with its installed MIME types, passes the arriving peer agent to the peer agent framework. (It is also possible to configure the peer agent framework, depending on the type of the peer agent, to run it immediately upon receipt on the local system without user intervention. This is sometimes useful, depending on the application.)

[2] Object serialization refers to the automatic writing out of a linked object structure to a data stream. Java allows the serialization of any object with the interface *Serializable* without additional programmer effort. Objects may later be *deserialized*, recreating the original object inside the Java Virtual Machine. Serialization is an easy way to transfer objects between computers.

```
public class UntrustedHandler implements Handler
{
    // The peer agent type with which this handler associates
    public String getPeerAgentType() { return "untrusted"; }
    // The expected class of peerAgentObject (argument to handlePeerAgent())
    public Class getDefiningClass()
        { return Class.forName("UntrustedPeerAgent"); }
    // Called to handle a peer agent
    public void handlePeerAgent(Object peerAgentObject)
        { ... }
}
public interface UntrustedPeerAgent
{
    public void doRun(UntrustedContext con);
}
public interface UntrustedContext
{
    public void sendReply(PeerAgent reply);
}
```

Figure 2: The handler and defining interface for the *untrusted* peer agent type.

2. The peer agent framework, upon receiving a peer agent, examines the peer agent's type and looks for an installed handler of the same type. If no handler is found, a *HandlerNotFound* exception is thrown and the process aborts.
3. Peer agent class files are made available to the Java Virtual Machine (JVM).
4. The peer agent object is deserialized.
5. The peer agent object is checked for being a subclass of the handler's defining class. If not, a *BadPeerAgentObjectClass* exception is thrown and the process aborts.
6. The handler is run on the peer agent object using the *handlePeerAgent()* method. (See Figure 1.)

Further verification and processing of the peer agent depends entirely on the peer agent's type and handler. Since handlers are often application-specific, that is beyond the scope of this paper. However, before running a peer agent, a handler will commonly take verification steps that include checking digital signatures and certificates, decrypting data, and setting up security policies and sandboxes.

Note that deserializing the peer agent object and invoking the handler on it could cause some or all of the peer agent classes to be loaded into the JVM and executed. (See Section 2.2.) This is a standard feature of Java's dynamic class loading mechanism and is the normal way to execute mobile code.

Peer agent types and handlers are application specific; however, one peer agent type and handler—*untrusted*—is a standard part of the peer agent framework. Developers may therefore create *untrusted* peer agents that will run on any peer agent installation.

2.1 Untrusted Peer Agent

The *untrusted* peer agent type (Figure 2) is used for peer agents for which no previous arrangement has been made between the sending and receiving party. Its handler executes the *doRun()* method of the peer agent object, which is an instance of a subclass of the defining class *UntrustedPeerAgent*.

No level of trust is assumed with the *UntrustedPeerAgent*. The handler therefore runs the peer agent under a completely restricted security policy designed to prevent the peer agent from effecting any change in the state of the recipient's system. No local or network I/O is allowed; however, as with Java applets,

certain kinds of GUI interfaces *are* allowed. This makes it possible for Alice to send Bob an elaborate graphical invitation.

The handler provides the *UntrustedPeerAgent* with one additional piece of functionality beyond the scope of the sandbox: the peer agent is allowed, pending user approval, to construct and send one (and only one) peer agent back to the stated sender of the original email. Email can be easily forged, and the stated sender is not necessarily the true sender. These restrictions are therefore put in place to prevent the peer agent from becoming a source of large-scale spam.

This functionality is provided in a flexible way through the use of a *context object* [10]—in this case defined by the *UntrustedContext* interface (Figure 2). The context object is created by the handler and passed to the peer agent. The peer agent is able to call methods in that object even if its sandboxing rules would otherwise prevent the actions taken by those methods. How to build secure context objects that cannot be abused by a peer agent to gain more capabilities than originally intended is described in [11]. Context objects therefore provide functionality similar to that of Safe-Tcl language extensions [3].

2.2 Java Classloading

As noted above, the act of deserializing a peer agent object and invoking the handler on it can cause some or all of the peer agent classes to be loaded into the JVM and executed. Java RMI [5, 7] popularized this method of passing and invoking mobile code in Java.

In Java RMI, classes are loaded on demand. When a class is needed during execution, the system contacts the source of the mobile code and requests the class. However, this model assumes network connectivity not generally available to peer agents. Peer agents can be executed off-line, for example.

The sender of a peer agent is therefore required to include with the peer agent object the Java bytecode for all "non-standard" classes that might be required to run that peer agent. By "non-standard," we mean classes that are not already assumed to be loaded on the recipient's machine because they are part of the Java system, the peer agent framework, or the particular handler associated with a peer agent's type.

Because Java classes are first-class objects within the Java system, determining this set of *required classes* to include with the peer agent object without the help of the object itself is undecidable. However, the vast majority of Java code loads classes in only a few standard ways that are readily apparent upon examination of an object's serialized form and its associated class files. This is sufficient, for all practical purposes, to allow the sender to automatically determine the set of classes required by a peer agent object [10]. In case the sender does not send the required set of classes, the resulting peer agent object will throw a *ClassNotFoundException* when run, aborting execution of the peer agent.

2.3 Peer Agent Applications

Having described the mechanics of how peer agents work, we will now proceed to show how they can be combined to form peer agent applications. A *peer agent application* consists not just of peer agents, but also of handlers and other locally installed software required to accomplish a specific task.

Figure 2.3 returns to our example and illustrates how Alice can use peer agents to plan her party. The steps of processing are indicated by circled numbers:

1. Alice installs the *MyParty* software application on her computer. She runs *MyParty*, entering her guest list into it.
2. *MyParty* generates and delivers invitation peer agents to Alice's guests. Since Alice has no particular prior arrangement with her guests, *MyParty* sends peer agents of type *untrusted*. Every peer agent is embedded with a random cookie, unique to each recipient.

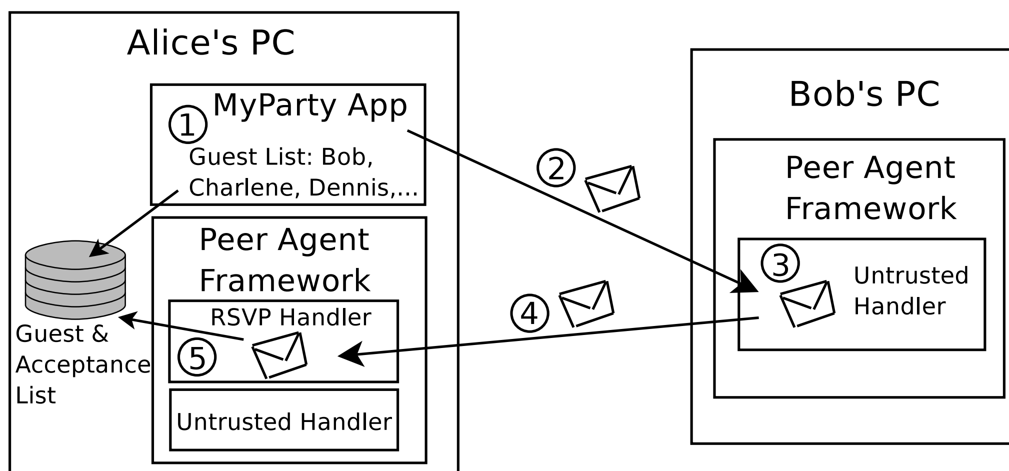

Figure 3: A round trip of a peer-agent session. Numbers correspond with steps in the text.

3. Bob receives and dispatches the invitation peer agent to the *untrusted* type handler, where it is executed. It displays an animated invitation and asks for his reply, which he enters.
4. The peer agent running on Bob's machine then returns a peer agent to Alice of peer agent type *RSVP*.
5. A custom handler on Alice's machine for peer agent type *RSVP*, installed as part of *MyParty*, processes the reply peer agent. The *RSVP* handler verifies that the reply really came from Bob. This is done by checking that a cookie returned in the *RSVP* peer agent matches the cookie originally sent to Bob. The RSVP peer agent is then allowed to update Bob's position in Alice's acceptance list.

This example highlights a few properties of peer agent applications:

- Only the initiating party needs to install specific software to use many peer agent application, including *MyParty*. This makes it easy for an initiator (Alice) to introduce new peer agent applications as the need arises without requiring her receiving parties to upgrade.

- The initiator of a peer agent application—Alice in this case—is analogous to the party that runs a web server in a traditional web application. In contrast to web applications, no special Internet service is required to serve as the initiator. Even devices with only email access can initiate peer agent interchanges.

- Peer agents do not necessarily need to involve executable code. A handler can certainly choose to extract data from its peer agent object without executing any code that came with it. This would be sufficient for the RSVP in the above example. Rather than using a context object to provide response list update methods to its peer agent, the *RSVP* handler could instead update Alice's response list directly with data extracted from the *RSVP* peer agent.

  Note that the fundamental trust issues do not change, whether or not peer agents involve executable code. If Alice is to trust the veracity of her response list, she must still protect herself against forged RSVPs from someone trying to impersonate Bob. In this case, a non-executable *RSVP*-type peer agent needs to have more privileges than an executable *untrusted* type peer agent for the simple reason that the *RSVP* peer agent must be able to update a bit in Alice's computer. Whether the handler or the peer agent ultimately runs the update code is immaterial.

- Up until now, we have assumed that handlers are fully trusted. This does not have to be the case. For example, Alice might not trust the *MyParty* software and might therefore wish to limit the system resources to which it has access. It is an easy thing with Java to run a handler in a sandbox with reduced privileges. Clearly, however, a handler cannot bestow privileges to a peer agent that it does not itself possess.

2.4 Secure Communication with Peer Agents

In the above peer agent application, Alice's *RSVP* handler takes steps to verify that the *RSVP* peer agent it receives is a valid response to an invitation that Alice sent. The cookie scheme described, while common for low-security web applications today, is vulnerable to many common threats.

In many cases, Alice might desire a higher level of authentication and security for her peer agent communication. One easy way to achieve this would be for her to send her peer agents using an encrypted, authenticated email protocol such as PGP mail [8]. Bob would then need to have a PGP-mail client installed in order to receive these peer agents.

It is also possible to build secure encryption and authentication directly into the peer agents and handlers themselves. This is done by building peer agents and handlers that implement suitable cryptographic protocols [4] over store-and-forward communication channels. Because of the large latency in such channels, protocols requiring few rounds of interaction are preferred.

Exactly how these peer agents would work and what threat models they assume is far outside the scope of this paper. Cryptographic systems are notoriously difficult to get right, requiring extensive review to verify their security. Suffice it to say that whatever cryptographic peer agent protocols might be developed, their implementation will be eased through the use of the Java cryptographic extensions, which are built into the standard Java library.

3 Future Work

In this paper, we have focused on the use of peer agents for a simple party planning application. Clearly, peer agents can be used for much more; in fact, they can serve as the basis of communication for any distributed application. We describe here just a few of the ways we see peer agents being used in the future.

3.1 Distributed Algorithms Deployment

There is no shortage of sophisticated distributed algorithms, many which see little use. Consider Paxos [16, 21], for example. Providing a fault-tolerant distributed shared state, this algorithm could be used to create a distributed serverless workgroup. The workgroup would share a CVS-type repository [15] that is replicated on every member's node. Members would modify this repository by running peer agents that implement the Paxos protocol. Details remain to be worked out.

This is just one example of an advanced distributed algorithm that could be implemented and *put into real use* without requiring that it be adopted by Internet committees or that complex system-level servers be installed on one or more of the machines that will use it. Peer agents show promise as an easy testbed for distributed algorithms research.

3.2 Mobile Platforms, Extreme Environments

Many Internet devices today experience sporadic network connectivity. These devices include not only the proliferating array of mobile devices on low-grade wireless networks, but increasingly computers of all

kinds in extreme settings: in remote villages [20], in Arctic exploration [13], on the Interplanetary Internet [22]. In all cases, email is the only practical link to the Internet due to intermittent connectivity, long latency (up to 24 hours) or both.

It has been noted before that mobile code is essential for these situations because it allows a vast reduction in the number of network round trips [6]. Peer agent-based systems will work just as well in these extreme environments as they do for well-connected systems, providing an unprecedented level of network utility in extreme environments. We expect the number of applications for peer agents to grow in the future because of this proliferation of semi-connected Internet devices and because peer agents provide a standard framework in which these applications can be built.

3.3 Large Distributed Applications, Small Clients

We imagine a distributed system consisting of a centralized server and mobile clients with intermittent network access. The server maintains a vast repository of algorithms, too large to fit on each client. Over the course of communication, some of these algorithms need to be run from the client. This kind of system is easily implemented using peer agents: algorithms can be sent to the clients as needed, to be discarded after they have been run.

We expect this kind of system to become increasingly important as traditional client-server desktop systems are being transformed into distributed applications that communicate with semi-connected mobile clients. Hospital management systems are just one example: increasingly, medical professionals are being issued PDAs, which they use as their main interface to the enterprise-scale system provided by their hospital.

The emergence of wireless sensor networks almost requires this kind of approach [17] if the sensor nodes are to be used in any general-purpose fashion, since the sensor nodes are so tiny. It is not yet clear how one would sandbox peer agents on a tiny sensor node.

3.4 Conference Paper Submission

Peer agents could be used to replace the web applications currently used to submit papers to conferences. In its call for papers, a conference could distribute a peer agent used for submission. Researchers would store this peer agent in their *Inboxes*. When they wish to submit, they would run the peer agent and follow the instructions. It would check the submission and send an encrypted reply to the conference chair, whose locally installed software would process the submission appropriately. The researcher would retain the reply peer agent as evidence of what was submitted; the conference could send a second peer agent confirming that. Needless to say, this system would vastly reduce conference headaches as well as almost completely eliminate the need for a separate email submission option for researchers without adequate web access.

4 Conclusion

We present a system for object-oriented typed email. Parties communicate by exchanging typed executable email components known as *peer agents*. When processed at a receiving system, peer agents are dispatched by user-installed *handlers* based on their type. The action of a handler on a peer agent is entirely application-dependent. However, one handler comes built into every peer agent installation that runs its peer agent in a restrictive sandbox, assuming nothing about the sender.

Because they can be transported over store-and-forward networks and because interesting computations can be achieved using few round trips, just about any machine with email access can act as a “server” (initiator) or “client” (recipient) for a peer agent-based application. Furthermore, the executable nature of

peer agents allows the adoption of new communication standards even if only the initiating party chooses to adopt it; this solves a common chicken-and-egg problem.

We have implemented the basic peer agent framework in Java but have not yet integrated it with email clients and MIME types. Doing so should be straightforward and will give us a testbed in which to try out real applications such as the *MyParty* described here. Our implementation is available at `http://www.peeragent.org`.

Peer agents, as a general form of communication, can be used to implement any distributed application. Depending on the handler, they can communicate using all common security and authentication protocols. We expect peer agents will be particularly useful as a platform for distributed algorithm research and development, as a way to improve effective network connectivity for distributed applications in extreme environments, as a basis for a new class of distributed systems involving tiny clients, and in many cases as a convenient alternative to today's web applications.

References

[1] R. H. Anderson and J. J. Gillogly. "Rand Intelligent Terminal Agent (RITA): Design Philosophy". Technical Report R-1809-ARPA, Rand Corporation, February 1976.

[2] Nathanial S. Borenstein. "Computational mail as network infrastructure for computer-supported cooperative work". In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 67–74. ACM Press, 1992.

[3] Nathaniel S. Borenstein. "Email With a Mind of Its Own: The Safe-Tcl Language for Enabled Mail". In *Proceedings of the IFIP TC6/WG6.5 International Conference on Upper Layer Protocols, Architectures and Applications*, pages 389–402. Elsevier Science Inc., 1994.

[4] Steve Burnett and Stephen Paine. *RSA Security's Official Guide to Cryptogrpahy*. McGraw-Hill, 2001.

[5] Mary Campione and Kathy Walrath. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet*. Addison-Wesley Pub Co, 1998. `http://java.sun.com/docs/books/tutorial/index.html`.

[6] David Chess, Colin Harrison, and Aaron Kershenbaum. "Mobile Agents: Are They a Good Idea? – Update". In *Mobile Object Systems: Towards the Programmable Internet*, pages 46–48. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[7] Troy Bryan Downing. *Java RMI Remote Method Invocation*. IDG Books Worldwide, 1998.

[8] M. Elkins. *MIME Security with Pretty Good Privacy (PGP)*, 1996. `http://www.faqs.org/rfcs`.

[9] Robert Fischer. Peer agent web site. `http://www.peeragent.org`.

[10] Robert Fischer. "Chapter 5: Server Framework". In *Web Applications with Client-Side Storage, Harvard University Ph.D. Thesis*, pages 113–131, June 2003.

[11] Robert Fischer. "Chapter 8: Browser Implementation". In *Web Applications with Client-Side Storage, Harvard University Ph.D. Thesis*, pages 132–163, June 2003.

[12] N. Freed and N. Borenstein. *Internet RFC 2045–2049: Multipurpose Internet Mail Extensions (MIME) Parts 1–5*, 1996. `http://www.faqs.org/rfcs`.

[13] Fujitsu. “Achieving the extraordinary with mobile data on the Fujitsu Polar Challenge”. *uk.fujitsu.com News Archive*, April 2004. `http://uk.fujitsu.com/news/newsarchive/2004/04/200404`.

[14] Trent Jaeger and Atul Prakash. Support for the file system security requirements of computational e-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 1–9. ACM Press, 1994.

[15] Ralph Krause. CVS: an introduction. *Linux J.*, 2001(87):3, 2001.

[16] Leslie Lamport. “The part-time parliament”. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[17] Philip Levis, Neil Patel, David Culler, and Scott Shenker. “Trickle: A Self-Regulating Algorithm for Code Propogation and maintenance in Wireless Sensor Network”. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[18] Rajan M. Lukose, Eytan Adar, Joshua R. Tyler, and Caesar Sengupta. Shock: communicating with computational messages and automatic private profiles. In *Proceedings of the twelfth international conference on World Wide Web*, pages 291–300. ACM Press, 2003.

[19] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. “The Safe-Tcl Security Model”. In *Mobile Agents and Security*, pages 217–234, 1998.

[20] Alex (Sandy) Pentland, Richard Fletcher, and Amir Hasson. “DakNet: Rethinking Connectivity in Developing Nations”. *Computer*, 37(1):78–83, 2004.

[21] Sergio Rajsbaum. ACM SIGACT news distributed computing column 5. *SIGACT News*, 32(4):34–58, 2001.

[22] Website. *Interplanetary Internet Special Interest Group (IPNSIG)*. `http://www.ipnsig.org`.