Version of paper published in Computer Communications

A Mailbox Ownership Based Mechanism for Curbing Spam

Nadeem Jamali [a,*], Hongxing Geng [b]

[a] *Department of Computer Science, University of Saskatchewan, 176 Thorvaldson Building, 110 Science Place, Saskatoon, SK, S7N 5C9, Canada*

[b] *School of Computing & Information Systems, Athabasca University, 1 University Drive, Athabasca, AB, T9S 3A3, Canada*

Abstract

Not only does Spam put strain on computational and network resources, it also drains the attention resource of users. Existing approaches to combat spam make decisions on per sender or per message basis. Where the former demands explicit intervention by users, the latter suffers from false negatives and positives.

We take an owned-resource view of email spam. An email recipient's attention space as well as computational and network resources devoted to handling email are viewed as precious owned resources. Anyone interested in using these resources – by sending a message which must be delivered and stored on a server and subsequently viewed in a list of received messages, and possibly read – must purchase the resource in the form of sending rights. These rights may be to resources owned by individuals and groups, and may also be owned by individuals or group.

Our approach is based on the CyberOrgs model for encapsulating distributed owned resources for multi-agent computations. The model uses a small set of primitive operations to enable multi-agent computations to engage in a rich set of interactions in a market of owned resources.

We have developed key mechanisms to enable message senders and recipients to negotiate contracts for mailbox access, as well as to construct policies leading to implicit and automatic negotiations for typical cases. A prototype implementation is described, and examples of policies are presented. Experimental results show that system performance is policy-dependent: as long as policies are carefully designed, negotiation overhead is minimal.

Key words: Email, Spam, Ownership, CyberOrgs, Agents, Attention Space

* Corresponding author.
Email addresses: jamali@cs.usask.ca (Nadeem Jamali), billg@athabascau.ca (Hongxing Geng).

1 Introduction

Email has brought many benefits to users; however unsolicited or unwanted bulk email creates an onerous burden for both legitimate users and Email Service Providers (ESPs), which include consumer Internet Service Providers (ISPs) (such as AOL and Telus), free ESPs (such as Gmail and Hotmail), and organizations (such as governments, universities, and companies). Rampant unsolicited bulk email affects every Internet user to the extent that email has become an unreliable communication mechanism, and Internet users are losing their trust on email even the Internet [**?**].

There are various definitions of spam [**? ? ?**]. We use the term spam to refer to any *unsolicited messages*.

Spam has brought and continues to bring significant cost to email users as well as ESPs. It is estimated that spam accounts for 85% email traffic today and this percentage will be up to 90% at the end of 2007 if the current increase trend continues [**?**]. Spam cost business 50 billion dollars around the world in 2005 [**?**]. Not only does spam demand significant time and attention from legitimate individual users, but it also affects ESPs. Phishing email [**?**] attempts to deceive recipients into disclosing their confidential information, such as private bank information, credit card number, and social insurance number. Experts estimate that 50% of phishing email deceives Internet users by spoofing eBay and Paypal [**?**]. Finally, spammers, along with hackers and virus writers, usually disperse spam by compromising inadequately protected computers. Today, more than 80% of spam worldwide is sent by remote-controlled zombie PCs [**?**].

Spam brings significant cost to ESPs as well. An ESP may lose its reputation because spammers find it is easier and cheaper to use its email services to send spam [**?**]. ESPs are forced to invest greater resources, such as network bandwidth, antispam technologies, storage servers, and customer support, etc. in efforts to fight spam. For example, ESPs have to invest more effort on inhibiting incoming spam and dealing with customers' complaints. Last, but not the least, ESPs may suffer from being added to black lists [**?**] if spammers initiate a large numbers of spam through their email servers.

Incentives have been the "cornerstone of human existence" [**?**]. The email infrastructure allows sending email for free, which allows malicious spammers to advertise their products through email with no cost to them. All that spammers have to pay for is the use of a computer and a subscription to access the Internet. Compared to the large volume of spam they send, the marginal cost is negligible. It is estimated that spammers can make profit even if only one recipient out of every 100,000 purchases their advertised products [**?**].

Current email infrastructure uses SMTP (Simple Mail Transfer Protocol) [**?**] to

transfer email among users. SMTP has some inherent shortcomings. First, it does not sufficiently constrain senders, allowing them to transmit as many messages as they want for free. Second, senders do not need to stay online after they click the *send* button. Third, senders can modify the *Return-Path*, a header in the SMTP, which allows spammer to appear to be legitimate senders, but receive replies to the messages to their own addresses. SMTP has no authentication. The email infrastructure does nothing to protect users from unwanted messages except to delete them when they arrive.

A number of approaches have been used for combatting spam. Spam filters [**?** **?** **?** **?**] usually work at the recipient side by allowing recipients to block incoming spam. Signature-based approaches [**?** **?** **?** **?**] authenticate senders through allowing recipients to check additional headers inserted by senders. DiffMail [**?**] and TCP damping [**?**] work at the recipient side as well. None of these approaches involve the sending side which actually knows what the message actually contains and whether it will be welcome.

Another approach to controlling spam is to discourage spammers from spamming by making sending spam expensive. Market-based solutions, assigning costs in terms of micropayment, human interaction, or computational cost [**?** **?** **?** **?** **?**] are based on this idea. These approaches require senders to invest some effort in sending messages. However, these solutions are not flexible. To give a typical example, Zmail [**?**] requires senders to pay a small amount of money (e-pennies) when they deliver a message. All that a recipient can do is set the price for sending a message. Not only are these approaches very rigid, they are are systemic in their focus.

Where statistical approaches treat spam at the level of individual messages, non-statistical approaches such as black / white lists create rigid permanent relationships between communicating parties. What is missing is the facility for building long-term relationships with the flexibility matching the richness of interaction possibilities between the parties.

We present an ownership-based approach for spam control. We think of a mailbox as a type of precious resource, which others cannot consume freely. Anyone interested in consuming a mailbox resource must obtain consent of its owner. Senders can obtain consent by negotiating with the potential recipients. The outcome of a negotiation is a *contract*, which is a written consent regulating the privileges and responsibilities of both the sender and the recipient. The generated contract controls subsequent communication between the sender and the recipient. By employing contracts, we can combat spam not only on the recipient side but also on the sender side. Meanwhile, contracts also construct a long-term relationship between senders and recipients.

The contributions of this work include an ownership-based approach for controlling access to mailbox resources, implementation of a prototype, separation of policy

from mechanism, introduction of *group* ownership rights, identifying interesting scenarios, implementing example policies, and finally evaluation of feasibility of the approach using experiments.

The rest of this paper is organized as follows. Section 2 describes our ownership-based approach for message admission control. Prototype design and implementation are elaborated in Section 3. Section 4 presents analysis and experimental results. Finally, conclusion and future work are discussed.

2 Ownership-based Message Admission Control

Consider an email user Alice. She has an email client running on her laptop, which she uses to access multiple accounts. She has an open source mail server running on the laptop. So long as she has connectivity, she can send and receive messages directly. Let us call this account P. She has an email account C_A through her employer, company A. She also has free Hotmail account F_H. Now, if she were to change employers to work with company B, C_A would be deactivated, but an C_B may come into existence. Neither her personal account P nor F_H are effected by this change. If she were to add another free account F_G with Gmail, none of the other accounts (including F_H) are effected. Any of the accounts can be individually closed.

There are a number of accounts in this example, all owned by Alice, but each with a slightly different degree of ownership. P appears to represent the strongest type of ownership, so long as Alice uses the account within the limits put in place by her broadband service provider. How much mail she sends using the server is up to her; how much she receives isn't necessarily. C_A and C_B are owned by Alice in her role as an employee of the respective companies. Alice's ability to send email using these accounts is constrained by the constraints on the company's email. Too many messages being sent from the company's servers can take it over what is allowed by its own broadband provider, or have it blacklisted by recipients. The company may therefore limit individual users to some quota. Receiving of emails too may be constrained. There may also be the question of whether Alice is allowed to read messages sent to the company. ESPs like Gmail and Hotmail are about the same as the company, except that because the users are not their employees, there is no question about whether they may be allowed to read email directed at the companies. The ESPs distribute email sending and receiving privileges to users with contractual obligations. Negotiating all these types of ownership is complex.

The approach we take is to use the CyberOrgs [**?**] model for resource bounded multi-agent systems. A cyberorg encapsulates an amount of resource and a collection of multi-agent computations which may use the resource. Cyberorgs trade with other cyberorgs in resources using *eCash*: they negotiate *contracts* which stip-

ulate what *types* and *quantities* of resources will be available to them, and at what costs. The model has three key primitive operations: `isolate` creates a new cyberorg hosting a part of the creating cyberorg's computations as well as resources; `assimilate` allows a cyberorg to relinquish dedicated control of its computations and resources to another cyberorg; `migrate` allows a cyberorg to migrate from one cyberorg to another.

The ownership-based message admission control mechanism is built by specializing the CyberOrgs model. The agents encapsulated in a cyberorg are email agents; the resource is a mailbox. A mailbox abstracts over physical entities, such as hard drive storage, CPU cycles, and network bandwidth, as well as the time and attention its owner devotes to reading email.

We use a special type of cyberorgs called *communicators*. There are two basic types of communicators: cells and groups. Communication among communicators is through sending and receiving messages. A communicator can join a group communicator and then use the group's resources. A cell can have multiple parent groups. If a cell already belongs to a group, to join another group, it creates a proxy, which enters the group and represents the creating cell. The proxy is called a *satellite*, which is the third type of communicator. A group can only belong to one group.

There are two types of agents inside a communicator: *user interface* (UI) agents which act on behalf of users, and *mail handler* (MH) agents, which handle email from and to the user.

A sender requires *write* permission to a recipient's mailbox before it sends messages to the recipient. To obtain *write* permission, a sender initiates a negotiation process with the recipient by sending a proposal offering a payment in *e-pennies*, which abstract over computational cost, human effort, or real money. For the sender, the desired outcome of the negotiation process is a *contract*, which stipulates the terms for subsequent communication between the sender and the recipient. Whether a contract is reached depends on both the sender's and the recipient's *policies*.

Example: Representing Alice's email

Returning to Alice's example, using this approach, companies A and B, as well as ESPs Gmail and Hotmail would all be represented by group communicators. Alice would be represented by a cell for the account P on her laptop. She would begin with satellites for C_A and F_H. Each of these satellites will be limited in their capabilities according to Alice's contracts with A and Hotmail. When alice moves to company B, her satellite at A can move to B. To begin a new free email service with Gmail, she creates a new satellite and places it in the group for Gmail, where it is permitted to use email according to the contract with Gmail. □

2.1 Types of communicators

Three types of communicators are used: cells, groups, and satellites.

2.1.1 Cells

A *cell* abstracts over email user agents, which are used to send and receive messages. A *cell* is a communicator which encapsulates one mailbox and one user interface agent, which provides an interface for sending or receiving messages or for management purposes (such as deleting messages and creating new message folders) to human users. Figure 1 depicts a cell. The black dot inside the cell is the user interface agent. Notice that there is a mailbox in the cell.

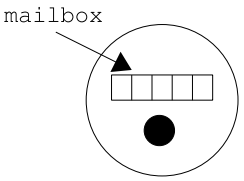

Fig. 1. Cell communicator

A mailbox is a resource which is controlled by the cell which encapsulates it. The encapsulated *agent* has *read*, *write*, and *delete* access right to the associated mailbox. Any incoming messages to the agent must be stored in the affiliated mailbox. Originators of messages who intend to deposit messages to a mailbox have to obtain *write* permission from the cell which encapsulates the mailbox. As long as a sender has a contract with a recipient, outgoing messages to the recipient will go to the mailbox of that recipient directly.

A cell can play various roles when it is under different circumstances.

- Sender: a cell which initiates a negotiation process for intending to send messages
- Recipient: a cell which examines proposals for receiving messages
- Subscriber: a cell which commences a negotiation process for intending to receive messages.
- Publisher: a cell which evaluates proposals for dispersing messages

In addition to the mailbox resource, a cell may have other resources, such as policies and contracts. Policies are used for negotiation. Contracts stipulate resource trade between two parties. We will discuss these two resources shortly.

A cell has a unique name but may have multiple *aliases* (corresponding to each of its satellites) and can use its name or its aliases when communicating with others.

2.1.2 Satellites

When a cell wants to join a group (described below), it creates a proxy, which is a duplicate of the creating cell. The proxy is called a *satellite*, which represents the creating cell in the hosting group. The name of the satellite is an alias of the creating cell. Before joining a group, the satellite must reach a contract with the potential hosting group. For each group that a cell wants to join, it creates a satellite.

A satellite has a unique name, always exists in a group, represents a cell in the hosting group, and can communicate with the creating cell. A satellite cannot create another satellite.

2.1.3 Groups

A *group* abstracts over an organization. A *group* is a communicator which contains an (incoming) mailbox, an (outgoing) message queue (MQ), a user interface agent and may encapsulate other communicating entities. The user interface agent is for communication and administration purposes. Each group member has a contract with its parent group, negotiated at the time of joining the group, which determines resource rights, such as access to the mailbox and MQ of the parent group. Members of a group can read messages in the group mailbox and can write messages to the MQ, as stipulated in their contracts. The group's user interface agent can read, write, and delete messages from the mailbox and the MQ.

If a group itself wants to send a message to a communicating entity, the message will be stored in the mailbox of the intended recipient provided there is an contract between them. If a group member wants to send a message to a recipient through its parent group, the message will be stored in the MQ of its parent group; whether the message is sent to the intended recipient or not is determined by the parent group.

A typical group is illustrated in Figure 2. The triangle is a satellite, which will be discussed shortly.

In addition to a mailbox and an outgoing message queue, a group may possess policies and contracts, which are resources that can be used by itself and its group members.

Like a cell, as a communicator, a group can be a sender, a recipient, a subscriber, or a publisher. Unlike a cell, a group cannot join multiple groups. In other words, a group can only have one parent group (if any).

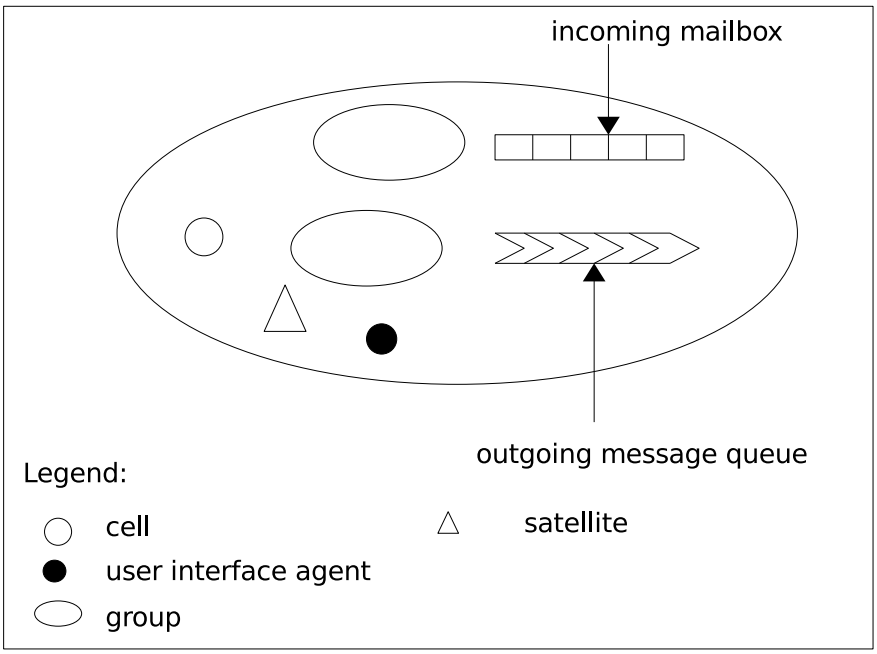

Fig. 2. Group communicator

2.2 Primitive Operations

A cell may decide to join or leave a group. Similarly, a group may join or leave its parent group.

2.2.1 Group Membership

A cell can be a member of multiple groups. A cell can join a group by creating a satellite and telling it to migrate to a particular group. The satellite tries to negotiate a contract to migrate to the group, and if successful, migrates.

When a cell decides to leave a group, it sends a message to the satellite representing it in the group. Upon receiving the message, the satellite migrates out of its parent group.

A group can only be a member of one group at a time, making group membership simpler. If a group wants to join another group, it tries to negotiate a contract with the target host, and if successful, migrates.

2.2.2 Policies

Each communicator maintains a policy pool, which consists of its sending policies and receiving policies. One selects a policy from its policy pool to meet its needs when it participates in a negotiating process.

For negotiation purposes, there are two types of policies which are *sending policies* and *receiving policies*. Sending policies are used by senders to make proposals. On the other side, receiving policies are used by recipients to evaluate proposals. Thus, to send messages, a sender must possess sending policies. Otherwise, it cannot interact with others. If a recipient does not want to evaluate proposals, it may not have receiving policies, meaning that the recipient is likely to permit all incoming messages to go through.

Policies can be arbitrarily complex. For example, a simple policy can be: *one e-penny for one message*. A more complex policy can look like: *ten e-pennies for ten messages within two days*, meaning that of up to ten messages may be sent to the recipient within 48 hours, during which the price of message sends will remain constant at ten e-pennies for ten messages.

A sender or a recipient may have general policies applicable to all receivers, as well as specific policies for a individual recipients. Similarly, recipients too may have general or sender-specific policies.

When a communicator is a member of a group, in addition to its own policies, it also has access to the parent group's policies. It can use these policies to make a proposal as a sender or evaluate a proposal as a recipient. A typical user, for example, would simply pick from the policies available through an employer's or an ESP's server. Because policies too are precious resources, whether or not a communicator can use its parent group's policies will depend on the terms of its contract.

A sender interested in sending messages to a recipient proposes a contract – based on its sending policies – to the recipient. The recipient responds to the proposal according to its receiving policies.

We use a simple two step protocol in which one party proposes a contract, and the other party either accepts or rejects the proposal. Although negotiation is not a focus of this work, because the overhead of our approach depends on the cost of negotiation, a complex negotiation protocol will not be suitable.

3 Prototype Design and Implementation

We built a prototype using Actor Architecture (AA) [**?**], which is an implementation of primitive agents called *actors* [**?**]. AA is a middleware system architecture, which provides an execution environment and an API to implement actors. An Actor execution environment on a computer node is called an AA platform. Each actor executes on an AA platform, which provides communication services to actors.

Because actors naturally model communicating concurrent processes, we use actors

to implement the communicators in our system.

3.1 System Architecture

There are three layers in our system architecture. From top to bottom, they are the user interface (UI) layer, the cyberorg layer, and the message handler (MH) layer, shown in Figure 3. At the sender side, messages flow from the UI layer to the MH layer, and at the recipient side, messages flow in the opposite direction.

The UI layer presents graphic user interfaces or command user interfaces to allow human users to exchange messages with others and perform management tasks, such as configuring policies and purchasing e-pennies.

The cyberorg layer hosts cells and groups. Cells and groups are communicating entities, which provide communication facilities and management facilities to the UI layer. In our prototype, communicators cannot communicate with each other directly. By contrast, each communicator must have an associated message handler, which is located in the MH layer. communicators interact with the external world through their affiliated message handlers.

The MH layer provides communication services to communicators in the cyberorg layer. Moreover, message handlers maintain information on behalf of associated communicators. For example, policies of a communicator are stored on its associated message handler so that the message handler can make a proposal or evaluate a proposal on behalf of the communicator automatically. Recall that cyberorgs use eCash to trade in resources; each communicator's eCash account is also maintained by its corresponding message handler.

Moreover, there is a Central Authority (CA) in the MH layer as well, whose purpose is to issue eCash and to ensure contract accountability. A copy of each contract is deposited with the CA for accountability purposes. CA is transparent to communicators. Only message handlers are aware of the existence of the CA.

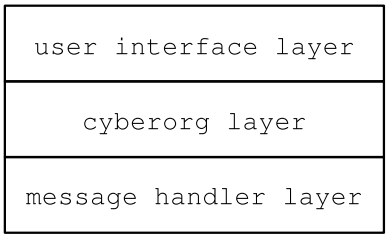

Fig. 3. System architecture: three layers

As shown in Figure 4, in our prototype, equivalent to email accounts, communicators are sources and targets of messages. Corresponding to real email servers, message handlers play the server role in our prototype. To simplify our discussion, we

refer to communicators as *clients* of their corresponding message handlers, which are *local* message handlers of the communicators.

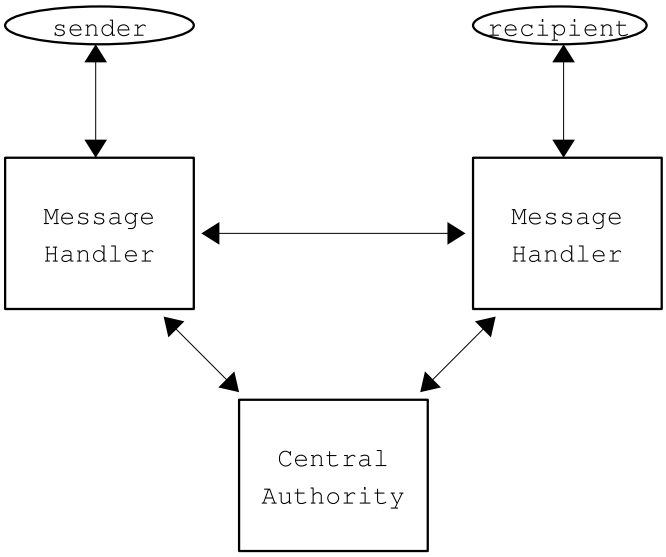

Fig. 4. Big picture of the prototype

A message handler runs on a computer node (or on an AA platform). However, one computer node may host multiple message handlers. Each message handler is a server which provides messaging services to its clients.

3.2 Communicators

Communicators, including cells and groups in the system, are implemented using actors. Besides communication and negotiation, communicators have capabilities to settle their sending and receiving policies, to join and depart a group, and to purchase e-pennies from their message handlers, etc.

3.2.1 Cells

A cell can be naturally implemented by an actor, which has a thread of execution and a message buffer to receive messages. Each cell must have an associated message handler to send and receive messages. To affiliate with a message handler, a cell must register itself with it by requesting the message handler to accept it. On receiving the registration request, the target message handler adds the requester to its client pool. Subsequently, the cell can communicate with the external world through its associated message handler.

In addition to sending and receiving messages, a cell needs to configure its policies, which are stored on its local message handler. On the one hand, as a message

sender, a cell can be a publisher. A publisher provides messaging services to others. On the other hand, as a message recipient, a cell can also be a subscriber. A subscriber subscribes to messaging services provided by publishers. In the publisher/subscriber interaction, the subscriber should initiate a negotiation process to receive messages.

If a cell runs out of its eCash, it has to purchase some from its local message handler. Otherwise, it can not send messages. When a cell decides to join or depart one of its parent groups, it sends a request to its local message handler, which, in turn, processes the request on behalf of the cell.

3.2.2 Groups

Like a cell, a group must affiliate with a message handler to interact with rest of the world. In order to associate with a message handler, a group must register itself just like a cell does. As communicators, groups also hold sending and receiving policies, which are kept on their local message handlers.

A group may permit cells or other groups to join it or may allow its members to leave it. In our implementation, these primitives are invoked using two APIs: `joinAGroup` and `departAGroup` exposed by the `Group` class. Because the members of a group vary over time, the group maintains a list to track them.

Group members can use resources of their parent group. In our implementation, resources that can be used directly are policies. In other words, a member can use policies of its parent group to make proposals or evaluate proposals.

A member can send messages through its parent group. Because resource consumption by group members is constrained as stipulated by their contracts with the parent group, the parent must be able to impose those constraints. For his, a group sets message quotas for its members, restricting the maximum number of messages that each member can send. A member may have a total quota as well as quota for particular recipients. If Alice, Bob and Cathy were members of the student, faculty and staff groups, contracts between the groups can be set so that Alice can send 10 messages to a recipient in the faculty group, but 20 for a recipient in the staff group. Hence, each group member may have multiple message quotas.

3.3 Central Authority

There is a *third party* called the central authority (CA) in the system. One purpose of the CA is to issue e-pennies. Only message handlers know of the existence of the CA, which is transparent to communicators. Communicators buy e-pennies from their affiliated message handlers, which, in turn, purchase e-pennies from the CA.

Another purpose of the CA is for contract accountability. When a contract is produced by a negotiation process, it is necessary to store a copy of the contract to the CA for the accountability purpose to prevent the recipient and the sender from denying their responsibility.

3.4 Message Handler

To interact with the external world, each communicator must have an associated message handler. Besides providing communication services, a message handler also is responsible for maintaining information for its clients, such as policies, contracts, eCash account, etc. Message handlers are implemented using actors.

3.4.1 Policy Pool

A message handler manages a policy pool, which contains sending and receiving policies of its clients. In addition to `adding` and `removing` policies from the pool, the most important task of the policy pool is to select a policy to make a proposal for senders or evaluate a proposal for recipients.

Upon accepting a message from its client, if the message does not contain a proposal, the local message handler generates a proposal for it automatically based on the client's policies. This task is delegated to its policy pool. If a recipient is identified, a policy specific to the recipient is preferred; if it does not exist, a general policy is used.

When a recipient-side message handler receives a proposal from a sender, it delegates the evaluation task to its policy pool, which looks for a policy specifically for the sender; if it does not exist, it picks a general policy.

Policies are of two types: *implicit* and *explicit*. Recipients or senders have implicit policies for handling individual incoming or outgoing messages. These policies lead to implicit negotiations followed by implicit contracts for individual messages with senders or recipients. Explicit policies are employed when a sender wants to have a long-term relationship with a recipient, so that every message does not require negotiation. These policies lead to explicit negotiations followed by explicit contracts for multiple messages with recipients.

3.4.2 Contract Pool

Recall that senders can send implicit proposals using implicit policies. An acceptable implicit proposal leads to an *implicit contract*, which is used only once. In our implementation, implicit contracts are not stored in the contract pool because they

are transient and only control one message communication. An *explicit contract* is generated by a negotiation using explicit policies. An explicit contract exists for a while to regulate subsequent communication between a sender and a recipient. In the following, when we say contracts, we mean explicit contracts.

3.4.3 Trigger Policies

In our prototype, communicators need to know their previous behaviors to determine how to interact with others in the future. Each communicator has its own policy for making decisions. Policies used to monitor the behavior of communicators are called *trigger* policies. An *event* is triggered once a certain condition is satisfied, which is defined in a trigger policy. For example, commonly, a sender may send messages to particular recipients regularly. It may be beneficial in this case for the sender to negotiate a long term contract with such recipients. Trigger policies of a communicator are maintained by its local message handler, which monitors behaviors of all clients. For example, a communicator may have a trigger policy such as: *if I have sent 10 messages to any user X in 24 hours, negotiate a contract with X, so that I may send them 20 messages per day.* If the communicator now sends more than 24 messages to Bob, the negotiation will be triggered by the message handler.

Each communicator has its own distinct trigger policies, which are kept on its associated message handler. A message handler may also have default trigger policies. Like sending and receiving policies, default trigger policies can be customized for particular communicators.

In our implementation, the message handler counts the number of messages that the monitored client has sent. If the number meets the defined number in the trigger policy, the message handler begins a negotiation on behalf of the monitored client.

3.5 Policy Selection

Recall that a communicator may have multiple sending and receiving policies. Conditions become more complex when a communicator is a member of a group, which may be a member of another group. A communicator can also use the policies of an ancestor group if permitted by its contract with its parent. The hierarchy of communicators brings a new problem: policy selection. For example, if Alice has her own sending and receiving policies, and she also belongs to the student group, which, in turn, is a member of the university group, which policies does she use in a particular situation.

In our implementation, the basic rule is that communicators prefer policies specific to other parties to general policies.

3.6 Policy Implementation

Our implementation separates policies from mechanisms. We expose a public policy interface, using which custom policies can be designed.

The public policy interface looks as follows:

```
public abstract class Policy implements
Serializable {
    int type;     // The type of policy
    int epennies; // Number of e-pennies

    // Each policy can make a proposal
    public abstract Prop makeProp();

    // Each policy can evaluate a proposal
    public abstract boolean evalProp(Prop p);
}
```

Each policy possesses two fields and two methods: `type` field describes the type of the policy, `epennies` specifies the number of e-pennies, method `makeProp` generates a proposal based on the policy, method `evalProp` evaluates acceptability of proposal.

Example: Implicit Policy

Implicit policies regulate individual message communications. In our implementation, an implicit policy describes the price of one message.

```
public class ImpPolicy extends Policy {

    public boolean evalProp(Proposal p) {
       ......
       int propEp = p.getEp();
       if  (propEp >= epennies)
            return true;

        return false;
    }

    public Prop makeProp() {
        ImpProp p = new ImpProp();
        p.setEp(epennies);
        return prop;
```

```
    }
}
```

Example: Message Policy

A message policy specifies that many e-pennies should be provided to compensate for a given number of messages. For example, *10 e-pennies for 10 messages* and *20 e-pennies for 10 messages* are message policies.

```
public class MsgPolicy extends Policy {
    int nrMsgs; // The number of messages

    public boolean checkProp(Prop p) {
       ......
       int propEp = p.getEp();
       int propMsgs = p.getNumMsgs();
       if  ((propEp/propMsgs) >=
                        (epennies/nrMsgs))
            return true;

        return false;
    }

    public Prop makeProp() {
        MsgProp p = new MsgProp();
        p.setEp(ep);
        p.setNumMsgs(nrMsgs);

        return p;
    }
}
```

4 Experimental Results

We carried out a number of experiments to assess the cost of using our approach. The design of our experiments is influenced by the consideration that approach is independent of the number of email users and servers. Although it is email users (individuals or groups) for whom policies are installed, the number of users does not directly influence the overhead. The overhead depends on the number of messages in the system, and the number of contracts that have to be negotiated. The processing required for ensuring that the contracts are adhered to is roughly the same for each message. Although different users may have different policies, re-

sulting in different frequencies of contract negotiations, we abstract over the users, and consider the frequencies of negotiations.

We also assume very simple negotiations, where one party proposes a contract and the other party either accepts it or rejects it. This does not effect the generality of our results because we simply count the number of contract negotiations. If a typical negotiation is more complex (and requires greater amount of interaction between the parties), the impact is larger.

Cost of system operations such as negotiation is something that either the system could absorb, or return to the parties responsible for the costs. For example, a party interested in commencing contract negotiations can be expected to pay for sending the proposal (directly or indirectly), thereby discouraging DoS attacks.

Our experiments focus on two key impacts of using the approach. First, we examine the increase in processing load on email servers, and consequently, effect on how many user messages can be handled by a server. This is important to assess whether a given server will be able to handle the expected number of messages, or to determine the configurations for a server capable of handling a certain number of messages.

Although the increase in processing demands on a server requires focus on the processor, message arrival delays are dominated by network delays. Network delays, in turn, are dominated by the frequency of contract negotiations carried out. Our second set of experiments show that a negotiation frequency lower than a certain threshold does not substantially effect the response time.

Experiments were carried out using two computers, each running an actor platform of AA. Each actor platform has one message handler and one cell. In our experiments, one cell as a sender delivers messages to another, which plays the recipient role. The sender runs on a laptop computer, which has an Intel P4 1.9 GHz CPU and 768MB memory. The recipient runs on a desktop computer, which has an Intel Core Duo 1.86 GHz CPU and 2GB memory. Both computers run Ubuntu Linux 7.04.

4.1 Processing Time

A number of experiments were carried out to collect data on the total processing time used between when the first message arrived and the last message departed departed. This was used in order to determine how our approach effected the total number of messages that a server may be able to handle.

As Figure 5 shows, how many messages a system can handle given an amount of processor time is only marginally impacted by our approach.

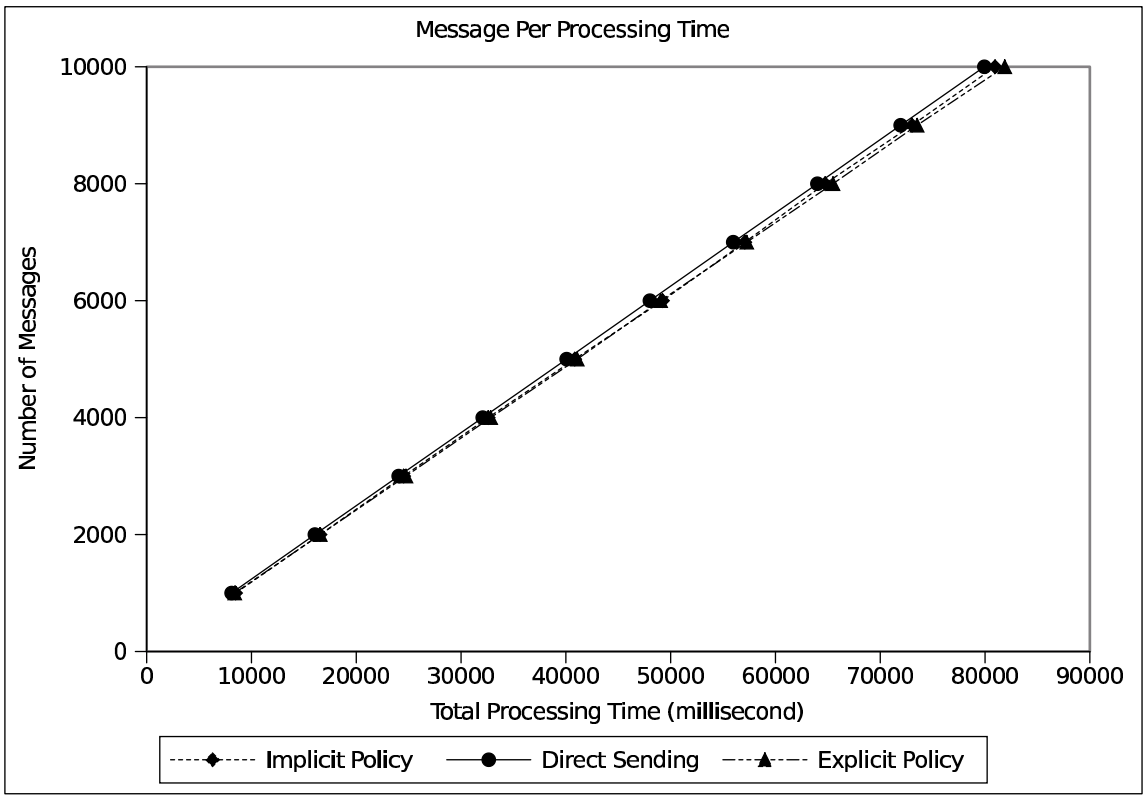

Fig. 5. Number of messages that a system can handle given an amount of processor time

4.2 Response Time

Figure 6 shows the relationship between the frequency of negotiation and the average response time, the time delay between when a message is ready for sending and when it is actually sent. The x-axis shows the frequency of negotiation. For example, *100* means a negotiation process is needed every 100 message sends. In each experiment, the sender sends 1,000 messages to the recipient. The figure shows that as the ratio of messages to contract increases, the response time of a negotiation system decreases.

Note that when a negotiation occurs every 25 messages or less frequently, response time of negotiation systems is stable and it asymptotically approaches the response time of the direct system. In other words, if an average explicit negotiation is about 25 messages or more, the response time is not much worse than that in a system not using our approach.

4.3 Interpreting Results

Although our experiments involved two user agents on two computers, because the experiments are focused on the number of negotiations and individual messages

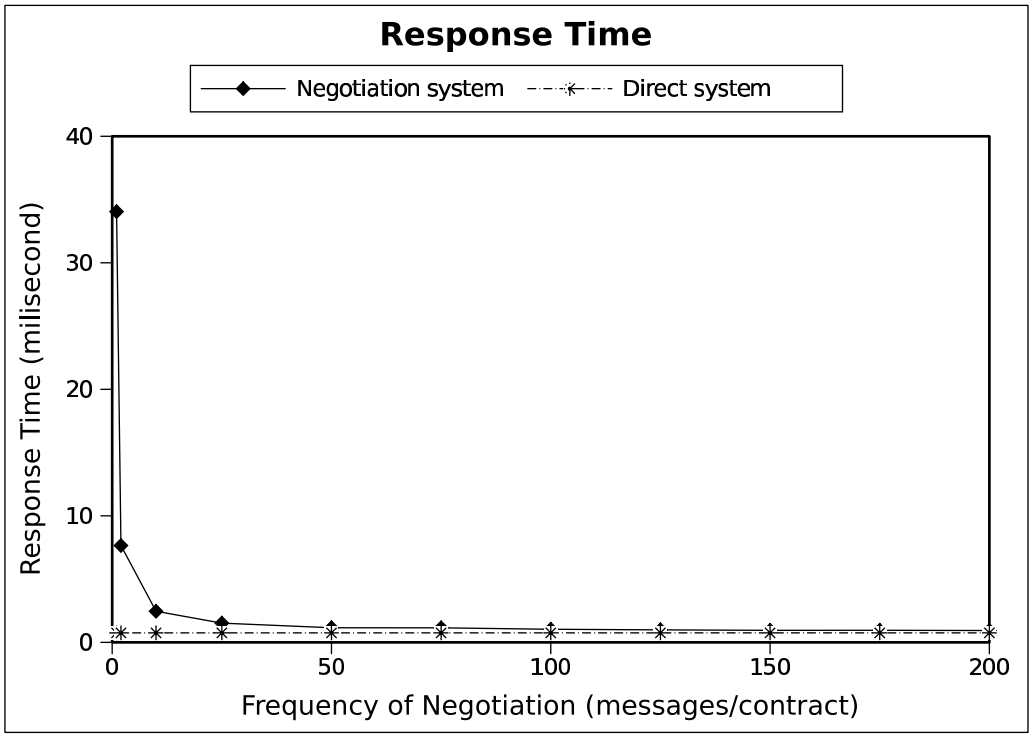

Fig. 6. The response time vs. frequency of negotiation

(neither of which are directly related to the number of users or computers), the results apply just as well to systems with larger numbers of users and computers.

A limitation of our experimental work is that simulated instead of real workload is used to examine our systems. Additionally, we use very simple negotiations; it will be interesting to see the impact of more complex negotiations, particularly in conjunction with incentives to reduce the complexity of negotiations.

5 Conclusion and Future Work

Curbing spam is a difficult challenge with a number of aspects. We have attempted to isolate the resource ownership aspect of the problem, and have developed mechanisms aimed at enabling long term relationships between communicating parties as diverse as individual email users, email service providers, and organizations. Users can customize inherited policies or write their own policies in order to automatically handle individual messages or to negotiate simple contracts stipulating terms for long term relationships.

A proof-of-concept prototype has been implemented. We have provided examples of how policies may be constructed. Our experiments show that if the number of contract negotiations is not very high, the effect of our approach on average response time as well as the number of message that a server can support is negli-

gible. Because the number of contract negotiations depends on the effectiveness of policies used, one can conclude that feasibility of the approach depends on the quality of policies used.

Clients based on this approach can be deployed incrementally, so that special policies handle communication with servers not equipped with compatible message handlers. The approach can also be readily used together with existing anti-spam solutions.

Further work is warranted in a number of directions. Our experimental work used a simulated message set; testing the approach using an actual email workload will be important. An orthogonal direction would be to carry out user studies to compare the user effort taken by approaches such as black / white listing or simply deleting unwanted messages, with maintaining policies required for managing email.

Acknowledgements

This work was supported by NSERC, Canadian Foundation for Innovation and the Government of Saskatchewan.

References

Author Bios

Nadeem Jamali

Dr. Nadeem Jamali is Assistant Professor of Computer Science at the University of Saskatchewan, Canada. His research interests include models, abstractions and languages for large-scale distributed and multi-agent systems. Dr. Jamali is a Kodak Fellow and a recipient of the Canadian Foundation for Innovations New Opportunities Fund Award. He was co-chair of International Workshops on Massively Multi-Agent Systems (MMAS 2006) and Coordination and Control in Massively Multi-Agent Systems (CCMMS 2007), and is a co-chair of CCMMS 2008. He received a Ph.D. from the University of Illinois at Urbana-Champaign, a Masters degree from Dalhousie University, and a Bachelors degree from the University of Karachi, all in Computer Science.

Hongxing Geng

Hongxing Geng is a Web Systems Analyst at the School of Information and Computing Systems, University of Athabasca, Canada. His research interests lie in the area of Distributed Systems. Hongxing Geng received a Master's degree in Computer Science from the Agents Laboratory at the University of Saskatchewan, and a Bachelor's degree in Computer Science from the Beijing Information Technology Institute. He currently works on mobile learning mechanism to support academic learning over mobile devices.