RSS/Email · v0.06

Overview

This paper discusses a new paradigm of email messaging ("RSS/Email") and contrasts it with the current paradigm ("Legacy Email"). In Legacy Email, senders push messages out to receivers, where they are aggregated by receiver MTAs and pulled by MUA software. In RSS/Email, senders are responsible for storing messages, and merely notify receiver aggregators that new mail is available; receiver MUAs then poll trusted senders for messages. Today (2005) we rely on email to perform many functions. This paper argues that by 2010 many, if not all, of those functions will be subsumed by a combination of Instant Messaging and RSS/Email.

Technology Implications. Next generation mail software will integrate RSS capability. Much messaging traffic will migrate to RSS/Email.

Business Implications. To be discussed.

Contents

Message Lifecycle in Brief · 3
Benefits · 4
Disadvantages · 6
Message Lifecycle in Detail · 7
How Composition Works · 8
How Submission Works · 9
How Transmission Works: Thundermouse · 10
How Notification Works: Thunderclap · 11
How Receiving Works: Thundercat · 13
How Reading Works: Thunderbird · 15
What About Spam? · 16
How Archiving Works · 17
How First Contact Works · 18
Essential Differences · 19
What does the Zombie Scenario look like? · 20
Implications: ISPs · 21
Implications: Enterprises · 22
Implications: Traditional End-Users · 23
Implications: Mobile Users · 24
Implications: Discussion Mailing List Providers · 25
Implications: Announcement Mailing List Providers · 26
Implications: Marketing Service Providers · 27
Implications: Blogs · 28
Efficiency Considerations · 29
Messaging Theory · 30
Push and Pull Models · 31
Time Sensitivity · 32
Audience Sizes · 33
Privacy Levels · 34
Threat Models · 35
Evolved Mechanisms · 36
Miscellaneous Considerations · 37
Practices: Today and Tomorrow · 38
How Entrenched is the Legacy Model? · 39
Managing User Expectations · 40
Transition Strategy · 41
Backward and Forward Compatibility · 42
Major Components · 43
Thunderbird Integrated MUA · 44
Thundercat Receiver and MDA · 45
Thundermouse Publisher and MSA · 46
Thunderclap Notification Protocol · 47
Integration Challenges · 48
FAQs · 49
Security Model · 50
Forwarding · 51
Acknowledgements ·

Message Lifecycle in Brief

Alice sends a mesage to Bob.

Alice composes the message in her client. She clicks "send".

Her client submits the message to a Message Submission Agent (MSA). This upload could occur over traditional SMTP on port 25, over SMTP on port 587, or using an HTTP POST using RSS conventions.

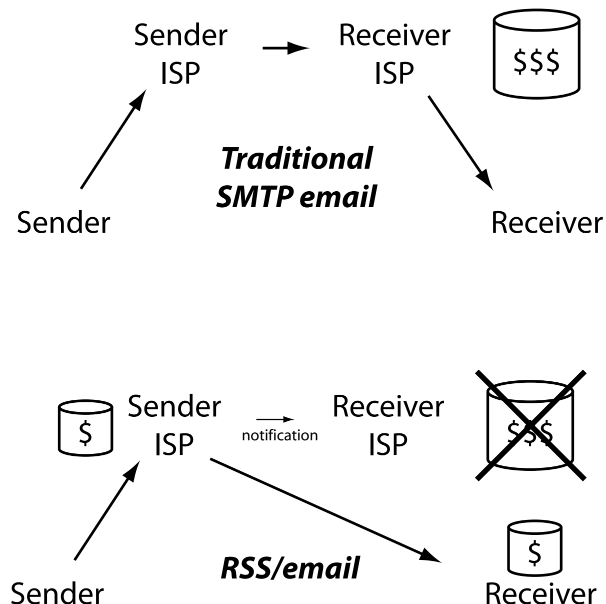
The MSA stores the message in the user's permanently-connected Outbox. The MSA observes that Bob is a receiver of that message. It establishes an directed RSS feed for Bob and only Bob to read that message.

The MSA sends a UDP notification to Bob's permanently-connected receiver ISP. That ISP records the notification in a database: "Alice has sent mail to Bob, and the feed URL is such-and-such."

Bob's messaging client (MUA) queries his ISP, and obtains the list of new mail as an RSS feed. Bob's MUA then polls each sender in the list, and pulls down the new messages. Alice is one of those senders, so Bob's MUA pulls down her message too.

Bob's MUA sorts the messages by arrival time or whatever. To Bob, the message inbox is indistinguishable from a traditional email inbox.

Bob's eyeballs read the message from Alice.



Benefits

What does RSS/Email do that Legacy Email doesn't?

Return Receipt. As a sender, you know exactly when the receiver's MUA pulled a message you sent. If the message was never pulled, you definitely know that the message didn't hit the eyeballs.

Retraction. If you go "oops, I shouldn't have sent that" you have a fighting chance to pull the message before the receiver gets it.

Dynamic Content. It's a lot easier to program dynamically generated content into an RSS feed than into an email message.

The Burden Shifts to Senders. A common refrain in the antispam community is "if only we could shift the cost of spam to the people who send it." In fact, the entire history of the antispam movement can be read as a succession of efforts to do exactly that, from postage stamps to hashcash to sender authentication to challenge response. The RSS/Email model requires senders to store messages, so the costs do naturally shift to the senders.

The Burden Shifts to the End-User. In fact, receiver ISPs don't have to store entire messages anymore; instead, they just keep a database of new mail, basically just a dirty list; the heavy lifting moves to the end-user MUA client, where CPU and bandwidth are cheaper. So RSS/Email conforms to the "dumb network, smart edge" model.

Bandwidth Replaces Disk. In Legacy Email, in the ideal case, a single message to 100 recipients at the same ISP goes across the network once, but repeats itself 100 times on disk. In RSS/Email, one message to 100 recipients at the same ISP goes across the network 100 times, but doesn't appear on

disk at all. In an era where bandwidth is cheaper than disk, this makes sense. (It would be really cool to have curves showing bandwidth vs disk over time, and maybe some kind of crossover. Consider the cost of transferring 1 gigabyte per hour, vs the cost of storing 1 gigabyte per hour. Kinetic vs static cost of data.

Of course a really smart mailserver implementation would store a message once and DB it a hundred times, but that shifts the big-O complexity to the programmer domain.

(True, we're still storing the message a hundred times on end-user disk, but that's a sunk cost anyway, invariant between Legacy Email and RSS/Email.)

No More "Over Quota" Bounces. The quota problem is an aftefact of receiver-side storage and the cost of ISP disk (static data). In a world where bandwidth (kinetic data) and end-user disk are cheap, it makes more sense to skip the ISP disk step entirely. And suddenly end-users are limited only by how much disk they have on their local cache devices, which nowadays is a lot.

Disadvantages

UDP notifications are unreliable. Messages may be delayed as a result.

Message Lifecycle in Detail

This chapter walks through every phase of a messaging transaction from fingers on keyboard to eyeballs on screen.

Terminology Note. When I say RSS, I mean it very loosely: Blogging with RSS is a pull paradigm which is getting more and more widely known, and it's the pull aspect of RSS that I'm interested in; whether the actual file format is XML or MIME is of less importance. More interesting is the fact that today Thunderbird supports both email and RSS in a single application.

How Composition Works

Alice opens a message composition box. This is ye olde textbox. It may appear inside a web browser, in the form of a webmail interface. It may appear inside an MUA, a traditional legacy email client. It may appear inside an IM chat window. It may appear inside a smart addressbook that supports integrated messaging.

I wouldn't be surprised if some enterprising OS vendor devised a smart messaging textbox widget that applications could simply embed. And then every application would suddenly become messaging-enabled, at least for publication.

Alice types her message into the textbox. The message could be a short "haha LOL" or a URL or a full-blown multiparagraph rant in HTML, complete with embedded images, video, and sound files.

How Submission Works

Alice has typed her message; she clicks “Send”.

The textbox closes. As far as Alice is concerned, the message has gone into the ether and it’s now The Internet’s job to make sure it gets to the receiver. The first step in this process is always the same: the underlying application submits the message to Alice’s ISP. (Or enterprise, or personal Linux server, or whatever. The important thing is that the MSA which gets the message is directly and permanently connected to the Internet.)

This submission could occur over plain old SMTP over port 25. Or it could occur over authenticated SMTP over port 587. These things happen if Alice is using a Legacy Email client that has no knowledge of RSS/Email. RSS/Email can be entirely backward-compatible with Legacy Email, at both the user-experience level and the technology-protocol level.

Or maybe the submission occurs over HTTP on port 80, or HTTPS on port 443. This could happen if Alice is using an RSS composition client like MarsEdit, so instead of clicking “Send” she’s clicking “Post” or “Publish”. Blog composition clients tend to use a Web Services protocol such as XML-RPC or SOAP to submit messages from the end-user client to the blog server. Those protocols use HTTP POST when they can. (Some blog clients also know how to use other protocols, like FTP, but let’s not go there.) Atom is another protocol in this space too.

But maybe Alice is typing into an RSS/Email aware client. In this next-generation scenario maybe there are two buttons: “Send Fast” and “Send Slow”. What’s the difference? “Send Fast” is what we associate with Instant Messaging. “Send Slow” is what we associate with Legacy Email. It tells the system whether Alice is impatient for a response.

In any case, let’s pretend the server that receives the submission groks RSS/Email. So it knows how to talk SMTP and it knows how to talk XML-RPC and SOAP too. It can handle submissions from Legacy Email clients and it can handle submissions from RSS/Email clients. It is super awesome. We call this MSA-on-steroids “Thundermouse”.

How Transmission Works: Thundermouse

Thundermouse accepts the message from the submitting client. It stores the message on disk in the user's online outbox. Once the message is in the outbox, it becomes accessible over HTTP by the recipient of the message. So if the recipient is expecting the message, and is impatiently pounding on the "Refresh" button, then once the message gets saved, it shows up in the receiver's client. So far this looks exactly like RSS blogging.

But we don't want to encourage people to pound on "Refresh" – we have quite enough déclassé behaviour in the real world and we don't need more of it online. That's why, after saving the message to disk, Thundermouse also sends out a UDP notification to the receiver. The receiver, if it is RSS/Email aware, accepts the notification and sends back a UDP confirmation to the sender. This polite little exchange, very lightweight, is enough to tell receivers "You've Got Mail!"

If Thundermouse gets back a confirmation, then its job is done. (To be precise, it needs to get back N confirmations for N recipients.)

But if it doesn't get back a confirmation, then maybe the recipient isn't RSS/Email aware, and so we have to fall back to backward compatibility: SMTP. Thundermouse dutifully queues the message for outbound SMTP and relies on the old messaging hindbrain to do its thing. Note that when it sends the message, it adds a header: X-MY-OTHER-FORMAT-IS-AN-RSS-FEED, with the URL to the RSS version of the message.

How Notification Works: Thunderclap

The UDP notification packet sent by Thundermouse contains a few important fields: the sender's email address, the receiver's email address, a unique message-id, and the feed URL for the one-to-one channel between them.

Maybe there's a variant, where if the data payload is short enough, the message itself gets bundled into the packet too, and there's a bit field in the UDP packet that says "this is a heavyweight notification". Oh, and there's another bitfield, too: whether the sender clicked "send fast" or "send slow". That tells us whether we're in IM mode or email mode.

We call this protocol "Thunderclap". (I thought about calling it "squeak" instead. Maybe we'll change the name to that.)

Where does Thundermouse send the notification? It does an SRV DNS lookup against the receiver's email address. If the receiver doesn't have an SRV Thunderclap record, then Thundermouse knows not to even try. But if the receiver does support Thunderclap notifications, then that's a signal to the sender that the receiver is RSS/Email aware. SRV records let the receiver specify servers which handle Thunderclap notifications.

The Thunderclap protocol supports confirmations: a receiver can ACK the notification with a UDP packet of its own. Confirmations are sent back to the IP address that originated the notification. They contain a bitfield that says "thanks, don't bother sending it via SMTP". Polite receivers should send back a confirmation whether they decline SMTP or not: sometimes the bit goes one way, sometimes the bit goes the other way.

How long should Thundermouse wait for a confirmation? We don't want to wait too long, because we want to fall back to SMTP quickly, and not hold up the works. But actually the SRV record gives us a clue: if there's no SRV record, we go straight to SMTP. If there is an SRV record, we wait 30 seconds; if no confirmation came back, we send another UDP notification. We try maybe twenty times in total, for 10

minutes, at 30 second intervals, on the theory that the typical reboot time for a Unix server shouldn't go much over that. If we don't get a confirmation after ten minutes, we just give up! If an SRV record was found, and we do not get a confirmation that declines SMTP, we do not attempt to send the message via SMTP. I know. Shocker.

Why is it okay to let UDP fail? UDP is not reliable, and servers go down; but we can face that with equanimity! In the brave new world of RSS/Email, it's ultimately the end-user's job to poll all their senders every so often. Yup, you heard that right. More on that later.

How Receiving Works: Thundercat

There are two parts to receiving: there's the ISP hub, and there's the actual end-user leaf. In this section we focus on the hub: the ISP, or the enterprise, or the Linux server that gets mail.

Traditionally, a mail hub has a cluster of MTAs bristling with all kinds of defenses, from DNSBLs to TCP throttling to challenge-response. In fact, a lot of receivers nowadays implement a defense-in-depth philosophy, with edge MTAs on the outside, UNIX MTAs in the middle, and antispam like Brightmail and antivirus like McAfee on the inside, like wagons circled around a pink and vulnerable Microsoft Exchange server. And maybe all those defenses are still there: in the RSS/Email world, for backward compatibility we might want to still accept messages over port 25. But the RSS/Email MTA-on-steroids also listens for UDP notifications. We call it Thundercat.

If Thundercat gets a UDP notification from a sender, it knows: "one of my users (maybe) has got new mail!" And so it records that assertion into a database. The database looks pretty darn simple: four fields, *timestamp* + *sender* + *receiver* + *feed_url* are enough to go on. And that database is exposed to receiver end-users in the form of … you guessed it, an RSS feed!

That works fine if the end-users are all RSS/Email aware. If Thundercat knows somehow that the end-users are doing the RSS/EMail thing, then it can send an SMTP-decline confirmation UDP packet back to the sender, so the sender doesn't try to follow up with a copy via SMTP.

Just how does Thundercat know which users are RSS/Email aware? That's an open problem: maybe it observes that some users are pulling down the dirty-list feed, and assumes that those users don't need an SMTP version of the message. Or maybe there's some kind of explicit configuration. But this doesn't seem too hard.

If the recipient of the message is not RSS/Email aware, though, then Thundercat doesn't send an SMTP-decline confirmation packet for that recipient. It sends a confirmation

packet all right, but the SMTP-decline bit is not set. So in effect it's saying "please do send that message over SMTP." And if the receiver site has an SMTP listener running, we just fall back to Legacy mode, and the message shows up over SMTP.

But hey! Maybe Thundercat's gotten so tired of all that SMTP nonsense that it takes matters into its own hands. Maybe the receiver thinks SMTP is fine for message submission, but not for over-the-net transmission anymore. If a receiving site has made the leap of faith to RSS/Email and is simply declining all incoming SMTP, it can still offer backward compatibility with its end-users, to a degree. What does it do? It sends back a confirmation packet with SMTP-decline set to "yes". And then it reaches out to the sender and it pulls down the message itself, and sends it straight into the local message store. The legacy end-user pulling mail down over POP or IMAP never knows the difference.

From lunch with Jarrod Roberson, Meng realized that it's possible to offer an RSS feed to the receiver client, even if there's no RSS happening anywhere else in the system; it just has to upgrade Legacy Email messages into RSS format. This lets people pull an RSS feed of new mail, which is something they want to do already. Symmetrically, a POP proxy can read RSS on one end and serve POP on the other – and this proxy can even live on the end-user's box!

How Reading Works: Thunderbird

In the legacy model, an MUA polls the ISP's server repeatedly. POP is what people know, and it's still what most people use. IMAP makes the constant polling a little less work, but the model is basically the same. Messages accumulate in the receiver ISP's mailstore, and are periodically downloaded by the receiver MUA.

In the RSS/Email world, the receiver's client asks their ISP not for a complete dump of all the new mail, but for a list of pointers to new mail. "Who sent me mail?" asks Thunderbird, and Thundercat replies with a list of senders and feed urls. Then Thunderbird goes off and polls each one of those senders directly. This model tremendously lessens the burden on the receiver ISP. No more POP, no more IMAP, no more accumulating new messages, no more spam, really. What a relief. I can't wait to get there.

But what if the UDP notification failed? And what if the sender has drunk so much RSS/Email Kool-Aid that they're not even falling back to SMTP any more? Then it's up to receivers to poll everyone in their addressbook. That's right: everyone who sends you mail, you ping them once a day to see if they have anything new for you. Is that model inefficient? Yes! Does the blogosphere care? No! Should we?

James Baldwin strongly disagrees with the design principle that notification should be allowed to be unreliable; he believes that the global polling is inefficient and should not be required ever. This places a higher burden of reliability on the notification protocol, and requires that a continuously-connected ISP always be in the picture.

What About Spam?

Spammers will adapt. They will queue messages in their outbound RSS feed boxes. They will send UDP notifications. Thundercat will accept those notifications. It may even respond with confirmations. And Thundercat will dutifully relay to you all the spam feeds mixed in with the regular mail feeds.

That's where the default-accept versus default-reject paradigm reversal kicks in. You need to make a basic decision: do you want to accept input from people who aren't in your addressbook? If you do not, you will not get spam, but you may have a harder time getting mail from strangers. If you do, you will get mail from strangers, but you will also get spam.

The difference is, if you make the decision not to get mail from strangers, Thunderbird will filter out all that spam without even bothering to download it. The total cost to the receiver is now one UDP packet of bandwidth, one row in a database, and one XML stanza to the end-user client.

How Archiving Works

Foo.

How First Contact Works

If you imagine a system which is default-reject, assume-guility-until-proven-innocent, whitelisting-only, where the addressbook is essential, and anybody who's not on your buddy list simply doesn't get through, the first contact problem becomes paramount.

Partly it's a question of expectations. In IM, many people expect that if you're not on the buddy list, you don't get through. In Email, people expect total strangers to write in.

The whitelisting-only, addressbook model is a wrenching transition for email. The end-user market looks like it really wants to move in that direction, but several obstacles stand in the way: it's hard for an ISP MTA to know what the end-user's MUA addressbook looks like, so it's hard to do a reject at SMTP time. But this where things are headed.

In an RSS model, your addressbook and your list of subscribed feeds are one, and so the whitelisting-only model is implicit.

In both cases the first contact problem arises.

There are several approaches to solving the problem. I believe that if we jam them together, they shore each other up and solve enough of the first contact problem to permit progress.

Note, first, that the first contact problem only applies to people you haven't already got in your addressbook. Everyone you already know is safe. Second, you can use domain reputation: if AOL users generally don't spam, anyone sending mail from AOL will get through. Third, you can use FOAF and Social Networking type solutions to investigate degrees of separation. Fourth, you can do challenge/response, as long as you don't think it's evil. Put together these things solve first contact well enough.

Essential Differences

At heart this is all about making it possible for receivers to only get mail from people they know. RSS/Email makes this possible at much lower cost than Legacy Email. There are two factors. One, it's hard for end-users to upload their addressbooks to their ISPs, so an ISP MTA doesn't easily know who's on the whitelist and who's not, so it has to accept the message and pass it on to the client to delete. Two, this is horribly inefficient, because by the time the stranger-or-not, spam-or-not decision is made, all the costs have been borne by all parties involved. RSS/Email reduces the burden to the absolute minimum, and makes it less important that ISP MTAs know who's in the end-user's addressbook.

Things could still go the other way. ISPs could attempt to gather up their end-users' addressbooks and keep track of them centrally. Improved synchronization tools make this a feasible future.

Even if you want to get mail from people you don't know, the RSS/Email model makes it much cheaper (for the receiver ISPs) to relay spam (and the odd first-contact message) to end-users; that UDP packet is much less work than an entire SMTP transaction.

What does the Zombie Scenario look like?

Abuse complaints come back from receiver end-users to the receiver ISP; the receiver ISP feeds that either to the sender ISP directly using a standardized abuse reporting format, or indirectly via a reputation system which senders check. The sender ISP can look inside the sender's outbox and confirm that it's all spam; and then it can nuke that outbox. The nuking can be automated based on thresholds. Once enough end-user receivers complain, and reputation systems take note of those complaints, receiver ISPs can automatically ignore that first-contact sender without bearing the cost of actually downloading and storing the mail.

Receiver ISPs can take advantage of the fact that they maintain a new-mail index feed to just run through with a DB operation and remove complained-about first-contact senders from their end-users' index feeds.

Implications: ISPs

Foo.

Implications: Enterprises

Foo.

Implications: Traditional End-Users

Foo.

Implications: Mobile Users

Foo.

Implications: Discussion Mailing List Providers

Foo.

Implications: Announcement Mailing List Providers

Foo.

Implications: Marketing Service Providers

Foo.

Implications: Blogs

Foo.

Efficiency Considerations

Pull systems are inefficient, because you have to poll everyone all the time on the off chance something's changed. The notification protocol helps mitigate this burden, but it's not perfect: receivers will still want to poll senders on a regular basis, like maybe once a day, and because most of the time those polls will come back negative, you can argue that that's wasted bandwidth. And that argument is correct: a push-based system makes more sense in an ideal world. But in the real world spammers take advantage of the push model and the receivers end up paying the price. If you look at the total cost of a push world with spam, and a pull world without spam, the pull model wins.

So on a macroeconomic level it's cheaper to do a pull model than a push model. But on a microeconomic level, senders aren't going to want to switch: right now they have it easy, and they're not going to want to bear new burdens. We need more thinking here.

Messaging Theory

Foo.

Push and Pull Models

Time Sensitivity

Audience Sizes

We can categorize messages by number of recipients.

Self. A message is only readable by the author; it's not really to anyone, it's just there.

One target. A message is directed to a single recipient.

Multiple targets. A message goes to more than one recipient. We want to be able to support traditional CC and BCC semantics.

Public. A message could be put out there for all to read, without restriction.

Nonself. As a special case, a person might want to send a message that they want everybody-except-so-and-so to read.

We can also categorize messages in other ways.

By topic. This leads to tagging, which is all the rage.

Privacy Levels

We originally called this "security levels", but "privacy" may be more accurate. Ideally, a sender should be able to set a privacy level on every message.

None. Messages are totally public.

Privacy Through Obscurity. If you don't tell anyone about your blog, nobody will read it. Maybe.

Soft privacy. If a receiver gives out the feed URL, anyone they give it to will be able to read all the mail from a given sender to that receiver. This corresponds to the capability key model. If the feed URL contains a long random string, it's basically a capability key, and more or less unguessable; but if I wanted to give away that mailbox, I just give away the URL.

Hard privacy. A feed could be protected using HTTP authentication, at a number of levels: basic auth just requires a username/password pair, while a full-on security model might require a client-side certificate and two-factor authentication with an RSAID and biometrics.

Threat Models

Zombies.

Spoofing.

Man in the middle attacks.

Interception.

Nigerian cybercafe attack. The best way to defeat a Turing test is to hire a human.

Gullible humans need walled gardens.

Mailbombing.

Listbombing.

Evolved Mechanisms

The email industry has evolved a number of mechanisms. Some of them are universal to all messaging; some of them are unique to email. Let's look at how those mechanisms transfer to RSS/Email.

Challenge Response.

Spam Filtering (by Content).

Spam Filtering (by Sender).

Virus Filtering.

New Mail Notification.

Pulling Messages. Remember Pointcast?

Port 25 Blocking.

Port 587 Submission. This becomes https submission.

Cryptography. Transport crypto can be trivially handled by https. True end-to-end crypto can be embedded using S/MIME type techniques.

Sender Authentication. This was absent from Legacy Email and so we had to spend years of our lives and much effort hacking it in. With a pull architecture, it's built in. URLs authenticate themselves and are, IDN and ebay.com@badguy.com attacks notwithstanding, not generally spoofable. If you want certificate-based authentication you have good old HTTPS.

Miscellaneous Considerations

Threading.

Return Receipts.

Directed Presence. One-to-one "you've got mail" maps neatly to directed presence.

Repudiation. (And nonrepudiation.)

Retraction.

Search based virtual folders?

Support for bookmarking.

Dynamic Content. This is going to be interesting, because some receiver ISPs will want to pull the content down and serve it to the receiver end-user in traditional ways. So the content stops being dynamic when the receiver ISP grabs it, at time of sending, when it's actually meant to be dynamic up until the time the receiver client grabs it, at time of eyeballing. You can get around this with iframes or AJAX, of course.

Urgency and Prioritization. (Thanks to Fred Fuller). There is way too much distraction and interruption in the workplace. If people task-switch less, they can get more done. Imagine if, in addition to "Send Fast" and "Send Slow", there's a "I need an answer: immediately / by end of work day / in 20 minutes / by X date". That would help the receiver's MUA integrate messaging into a to-do system, and decide whether to pop up an intrusive dialogue box or not.

Practices: Today and Tomorrow

Foo.

How Entrenched is the Legacy Model?

Maybe this is all Qwerty vs Dvorak, and even though a pull model is better, the push model, with all its flaws, is here to stay. We need to engineer a full-blown disruption with a killer app and smooth upgrade path and everything. I think we can make the jump if we ever get to a point where we say to end-users "if you want to accept mail from strangers, that'll be an extra $5 a month". Will anyone dare say that?

Retooling an entire email infrastructure is a lot of work. What are the benefits? Already today we can get no spam if we make a number of tradeoffs: we tell our ISPs who's in our addressbook; we only accept mail from senders who are whitelisted in that addressbook; we apply SMTP authentication; and we reject or challenge-response mail from everybody else. RSS/Email lets us do essentially the same things, only cheaper; but does the savings really justify a wholesale conversion?

Maybe the best way to find out is build the service and see what people do with it.

We can probably expect the marketing industry (ESPC) to respond quickly.

Managing User Expectations

How much are end-users going to have to think about this paradigm shift?

The blogging revolution has done a lot of the work already: it has paved the way for a pull model. The IM revolution has trained people to understand whitelisting based on buddy lists and addressbooks.

Transition Strategy: How Do We Get From Here to There?

Do we set a sunrise date?

Do we establish gateways to legacy mail, in the same way we had gateways from BITNET to the Internet? These gateways might leave a role for systems like Goodmail.com. The Transcended Sphere says: "send mail using RSS, or pay the cost of filtering it." It could say this to the senders, or it could say this to the end-user receivers.

We can use an X-MY-OTHER-FORMAT-IS-AN-RSS-FEED header to get Thunderbird to automatically subscribe to feeds, and ratchet senders and receivers towards the new model.

Backward and Forward Compatibility

Foo.

Major Components

Foo.

Thunderbird Integrated MUA

Foo.

Thundercat Receiver and MDA

Jarrod Roberson suggested a POP/IMAP proxy to help with receiver-side upgrade path.

Thundermouse Publisher and MSA

Foo.

Thunderclap Notification Protocol

This notification protocol is more or less directed presence. Today, presence protocols are crude: you're available, or idle, or away, to everyone. That corresponds to RSS feeds today being the same one-to-many communication to everyone. If we can layer one-to-one messaging on top of RSS, we can do one-to-one presence: we can tell one person "I'm available, and by the way I just sent you mail" and we can tell another person "sorry Mom, I'm away, but I did read your last message."

Integration Challenges

Foo.

FAQs

What if a UDP notification gets lost? Too bad. It's the client's job to poll everything in the addressbook.

Isn't it inefficient to poll everyone? Yes. It's much more elegant to use a push model, because in a push model, the system is quiet except when someone has just said something. In a pull model, receivers are constantly going "did you say something to me?" and senders are constantly saying "no, I didn't." When people are half deaf, they shout more.

Aren't you glossing over the fact that we can do whitelisting-only using the existing email paradigm? Indeed, that is what sender authentication is all about: it lets us whitelist senders reliably.

Can senders handle the load?

If a receiver ISP wants to support webmail, won't they have to store all the mail anyway? Argh, you're right. Is there a way around this? Can they construct a mailbox on the fly based on recorded notifications?

Security Model

Initial first-contact notification maybe goes over HTTP from the sender ISP to the receiver ISP. That notification contains a sender-index feed URL which is secured using a capability key model with a really long random string. That sender-index feed URL can be grabbed over HTTPS by the receiver client. Each sender-index feed contains permalinks to individual messages, which themselves have their own really long random URLs.

we need to backload RSS capability into an addressbook.
Meng Wong: in other words, given an email address, we need a way to intuit the RSS feed URL for that email address
Meng Wong: we need to invent a DNS SRV methodology and a URL convention, i think
Andrei Freeman: sort of a rssm://pobox.com/mengweng
Meng Wong: we want to support the long random string capability key approach, and we want to support a more predictable url scheme.
Meng Wong: right
Meng Wong: so, given an email address andrei@freeman.com, we do an SRV lookup on freeman.com for its RSS/Email service, and then we connect to any one of those machines, and request rssm://freeman.com/re1/index?sender=andrei@freeman.com&rcpt=mengwong@pobox.com
Andrei Freeman: right
Meng Wong: now, we'll probably get an http basic auth challenge back.
Meng Wong: i wonder if we can cheat and say if the request is coming from an authorized requestor machine for pobox.com, skip auth
Meng Wong: that bypasses the password thing completely.
Meng Wong: okay, so, we need to spec this, and we need to write a prototype implementation.

Forwarding

If you give a permalink to somebody else, they can read that message, and that message only. The sender-index feed remains unknown, unless a receiver should be so foolish as to give it away. This is the elegant model, but we can't count on it; so we can just layer forwarding as a new message.

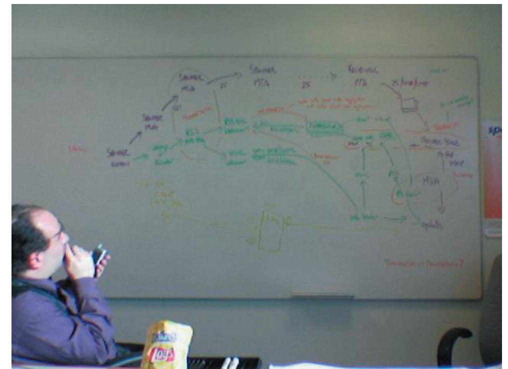
Acknowledgements

Much of this work was fleshed out during a discussion with Andrei Freeman on 20050404.

Andy Newton first brought the idea of RSS/Email to my attention at MAAWG in early 2005.

The original concept of sender-stores messaging was first publicized by DJB as IM2000. Pioneering message systems such as Zephyr established many of these concepts. http://ref.web.cern.ch/ref/CERN/CNL/2003/001/zephyr/

I am grateful for the feedback and discussion with David Mayne, Erik Fair, Robert Barclay, James Baldwin, and Juan Altmayer Pizzorno. Fred Fuller contributed several ideas regarding prioritization and urgency.



Appendix A: Thunderbird Patch Specification

Design a protocol that satisfies the following scenario:

Given a sender's email address, identify the server or servers which offer an RSS feed for that sender. A sender may offer multiple feeds: a public index feed may link to multiple one-to-many public blogs, one per topic; and there may be multiple private feeds, which are one-to-one. For example, given the address andrei@freeman.com, an SRV DNS query may identify one or more servers. You are at liberty to design a convention for public and private feed autodiscovery: an SRV DNS result may, for example, eventually lead to a query of the form http://rss01.freeman.com/re1/pub/index?username=andrei@freeman.com for the public feeds, and http://rss02.freeman.com/re1/pri/index?username=andrei@freeman.com&rcpt=mengwong@pobox.com for a one-to-one private feed. Backward compatibility with existing RSS conventions is preferred.

The sender and recipient may not have established a prior shared secret password; in that case, the sender may only be able to authenticate recipients according to IP address and, possibly, a client-side SSL certificate. For example, rss02.freeman.com may establish that the client IP for the HTTP query is a target of a (slightly differently-typed) SRV DNS query against pobox.com, and allow the pull; this security model is essentially equivalent to modern-day SMTP. Alternatively, for a higher level of security, rss02.freeman.com may require that the querying server offer a client certificate for pobox.com.

You must support alternative authentication models: if the recipient client pulls a unique per-message URL, or a per-edge (between sender and recipient) "channel" URL, without the benefit of SRV IP-based authentication or client-certificate crypto-based authentication, the server must also offer the message.