Smail: Using Solid for Email Storage and Sending

Chris Sarli
Brown University

Abstract

In this paper, we explore the feasibility and utility of implementing Solid protocol-oriented client and server applications for the purposes of writing, reading, sending, receiving, storing, and organizing email messages.

Many (if not the vast majority of) online services (and offline activities) "fall back" on email at some point, wether it be for service login verifications, detailed receipts, calendar invitations, etc. While it is likely that any given service an individual uses stores more potentially private information internally than it sends in emails, the fact that an individual's email account is effectively a cross-section of their digital life across *many* services and contexts make it a paradigm rife with privacy concerns. Although not universally true, many of the most popular email services do not expose messages storage directly.

We present three contributions: a Solid *ontology* and two applications capable of working with that ontology (one a client-side web application and the other a server-side SMTP server). These do not attempt to address all privacy vulnerabilities in email. Rather, we attempt to explore the possibility of cleanly disentangling the conveniences of modern email services from the underlying message storage, in an effort to increase transparency, user control, and interoperability between services.

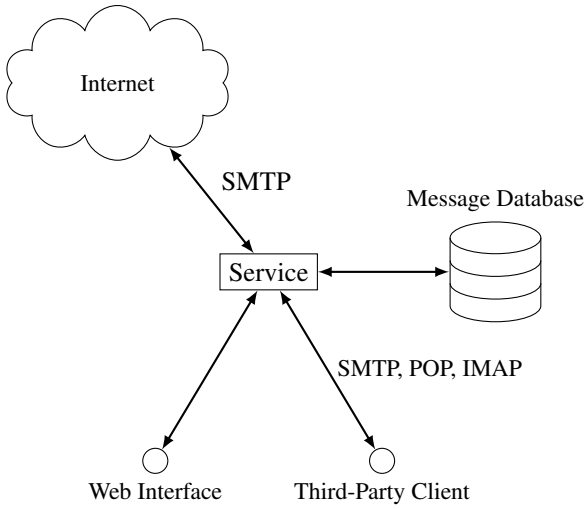

Figure 1: In the standard webmail model, the email service is central. Internally, a single email service may actually be implemented with many components (SMTP for sending and receiving messages, an API for the first-party web interface, database query logic, an IMAP server for third-party clients, etc.), but from an end-user's perspective, all interactions flow through a single service.

1 Introduction

With this work, we hope to address two problem areas: (1) privacy concerns arising from webmail service lock-in and (2) a lack of practical and useful applications of the Solid platform.

1.1 Webmail Concerns

The first motivation concerns the model that many popular email services (specifically, those services that can be called "Webmail" services [9]) use. In an effort the simplify the end-user experience, email providers typically configure and manage both the requisite servers needed for sending and receiving SMTP protocol messages and additional servers for storing received messages (and generally copies of messages previously sent by users as well) and associated metadata. Typically the email provider will offer a first-party web-based application for managing the user's mailbox and account, and may support third-party email clients through implementation of SMTP, IMAP, or POP syncing.

Figure 1 shows a highly simplified diagram of this model. Here, it is clear that users, accessing their emails through client applications, have no direct control over their stored

email messages, and must rely on the service not only to send and receive email, but also to view already-received messages. Should a previously-trusted service betray a user's confidences (for example, the service begins to mine years of stored messages for potentially valuable targeting information, as some have done in the past [10]) and the user wishes to move their mailbox to another service, the user must also rely upon that service to generate an export of its mailbox history and faithfully delete its copy of that potentially-valuable information. And, supposing that a user *is* able to generate an accurate export of their mailbox, email interchange formats generally lack user-generated metadata (read/unread status, labels, rules, filters, etc.) Although there is some hassle inherent in changing email service providers (domain name owners must update their MX records, but users of mass-market services such as Gmail must effectively start from scratch, as Google controls both the Gmail service and DNS records for `gmail.com` addresses), this additional hassle of exporting and storage concerns can be seen separately, and as an additional source of lock-in to a potentially privacy-invasive service.

Of course, there is nothing inherent to the design of email protocols that requires this model. Rather, webmail has grown popular *because* it provides a user-friendly abstraction on top of the configuration and maintenance of severs and DNS records, which can be unpleasant even for users who have the technical backgrounds such tasks require. There are likely product and profit motivations for webmail service providers to store messages internally (the aforementioned data-mining potential, but service-optimized indexing, compression, and security are other possible benefits of this model). The same service-dominant model of webmail is also found in many, if not most, modern web applications and services.

1.2 Solid

Solid [12] proposes a fairly radical departure from the aforementioned model, not just for email, but for most web applications and services. If the Solid protocol becomes widely supported across many services and applications, the overall Solid workflow may be refined to a point such that it becomes viable for widespread adoption. However, thus far, most Solid applications that have been developed are either purely demonstration utilities or games that offer no real benefit over existing offerings. There have been a number of attempts to build decentralized communication and social networking tools with Solid, but these are currently buggy to use in practice. They also suffer from a core limitation: they are built around Solid, and *require* users to connect a Solid Pod to use. This limitation is reasonable for those in Solid working groups or in a world where Solid is already popular. However, given that Solid is still very much an experimental and niche technology, any communication service that requires users to set up a Pod (not an altogether straightforward process) is at a disadvantage to the hundreds of alternatives that do not.

Email storage is a unique opportunity for Solid. Email is, for computing purposes, effectively universal. Though chat services have risen in popularity, email still remains a popular form of person-to-person communication, and the fact that so many online services fall back onto email for authentication and notification shows that it is still a useful and relevant tool. SMTP is very firmly entrenched, and so long as messages are sent using that protocol, then the mechanism of message storage at one end of a communication is of no concern or relevance to the correspondent. As a result, one individual can seamlessly begin using Solid for email storage without having to convince any of their regular correspondents to join them (thus avoiding the issue of slow/stalled adoption seen with client-side email message encryption). If email storage is shown to be a practical and robust use case for Solid, the tooling may improve, allowing for further development and the potential widespread adoption of the technology.

Given that Solid's model, along with the existing Pod software packages and development frameworks and tooling, is very much still experimental, another intent behind this work was to investigate challengers developers may face in working with Solid, and determining wether these are inherent to Solid's model or the product of nascent documentation and tooling.

2 Background/related work

Solid (Socially Linked Data) is an effort to unite a series of existing standards and protocols into a coherent vision for decentralized web applications and data storage. The Solid Protocol, still in draft form, outlines the use of technologies including WebID, WebSockets, and Resource Description Framework (RDF), that are used to create a Solid ecosystem [6]. In such a setting, users directly own and control Personal Online Datastores (Pods), which they can either host themselves or outsource to a Pod provider. In this model, statically-served client-side applications authenticate with a WebID (tracked partially with cookies) and receive permission to read from and write to a Pod via HTTP requests.

Email, of course, is ubiquitous and familiar. However, the concept of the "mailbox," what a user might think of a "their email account," is important to separate from that of the implementation of an email server, which may receive email for all users on a domain (or multiple domains, depending upon the configuration of DNS records). RFC 5322 clarifies this distinction: "A mailbox receives mail. It is a conceptual entity that does not necessarily pertain to file storage." [11]

A list of existing Solid Applications and use cases is available on the Solid Project's website [2]. In our research, it does not appear that there have been prior attempts to use Solid as a datastore for email messages.

Additionally, in our research we were unable to find previous attempts to specifically separate email storage from the server itself. There are several efforts to improve privacy

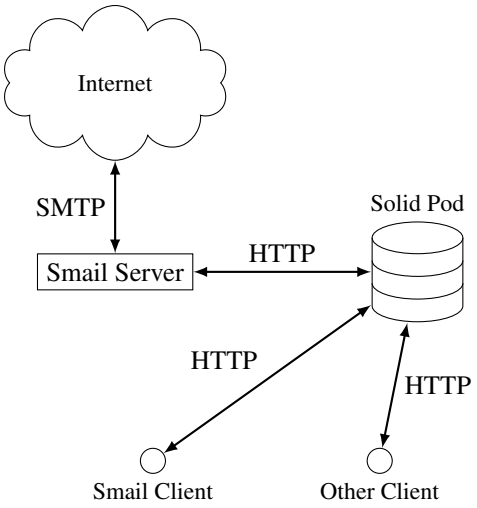

Figure 2: In the Smail model, the Solid Pod is central. This represents a major departure from the model shown in Figure 1, where the service (the closest analogy here is the Smail Server) is central.

and security in email (PGP and ProtonMail have focussed on adding encryption on top of existing protocols [16], new offering LedgerMail proposes replacing SMTP with a blockchain-based approach [7]), but we have found no record of attempts aligned directly with ours.

3 Design

In accordance with Solid's model, Smail (the working name for this project), proposes placing a Pod, rather than the email service itself, as the central component. (Shown in Figure 2.)

As discussed in the next section, Solid Pods can store structured data according to RDF specifications, called *ontologies* or *vocabularies*. These ontologies are typically publicly viewable. Preferably, these ontologies use inline RDF documentation, have internationalized names, and are accessible at an Internationalized Resource Identifier (IRI), but these preferences are not required and an ontology can easily be hosted on an individual's Solid Pod. For Smail, we have developed a simplified ontology for the storage of email, and have published in a Solid Pod at `https://chris-sarli.inrupt.net/voc/smail.ttl`.

To demonstrate the ability for reading, composing, sending, and recieving email, we have developed two additional applications. The first is a server ("Smail Server" in Figure 2) intended to run continuously (preferably in a datacenter) that implements standard SMTP mail handling, but which also saves messages to a specified Pod. The second is a client-side web application ("Smail Client" in Figure 2) that can be used to log into any Pod used for Smail and that presents an interface that mimics a familiar webmail experience. The "Other Client" in Figure 2 is not implemented; the figure merely includes this to indicate that any authorized application that can interpret the Smail ontology can interact with the system.

4 Implementation

This section is organized by the three major contributions: ontology, server, and client [13–15]. This implementation is **not** "fully featured;" a number of features one may expect from an email service (support for HTML email content/rich text, full access to message headers, etc.) have been omitted for simplicity, as their inclusion would merely inflate the size of the codebase and have little to do with measuring Solid's suitability. Additionally, while the ontology has been designed to allow for a single Pod being used for multiple users on the same (or different) domains, the server application presented here is designed as a demonstration for a single user on a single domain.

4.1 Ontology

The finalized ontology file is 74 lines long (most of which are annotations). The ontology defines the `Message` type, with the following attributes:

`messageId` A unique ID for each message, typically generated by the sender's mail client

`replyingTo` If the email is a reply, this is the ID of the direct predecessor message

`body` Plaintext content of email body

`subject` Plaintext content of email subject

`timestamp` Integer representation of timestamp

`from` Message sender

`to` Message recipient(s)

`is_read` Boolean indicating read/unread status of message

`directory` URL string of Smail directory message is currently filed in

In order to use this ontology easily in application code, Inrupt has release an `artifact-generator` tool, which consumes the ontology and produces target language-specific equivalents [8]. For this ontology, the tool generates a 183 line JavaScript (currently the only supported language) file that allows the developer to, after importing the exported `SMAIL` object, access ontology identifiers (URLs or IRIs) with simple dot-notation (for example, `SMAIL.messageId` can be used to retrieve content from a Pod instead of needing to hardcode an ontology URL into application code).

While this approach is helpful for describing data (and annotations exist to define domains and ranges on attributes, helpful in clarifying the intended data structure), ontologies are not strictly enforced. Similarly, while datasets (a file or object equivalent, with many attributes) are each identified with unique URLs, and can concretely exist within other datasets (called "containers"), Solid does not appear to offer any way of enforcing relations between datasets, and does not appear to have a clear mechanism to define expected container structures.

As a result, the directory structure that Smail uses on a user's Pod is not well-defined in RDF format in the way attributes are. That directory structure is defined as such:

To use Smail on a Pod, the user selects a base Smail container in that Pod. If we represent the "root" of the Pod as `/`, then this may be at `/smail/`. Inside this directory are two additional layers of organization before user mailboxes are represented. The first is domain names, and the second is user addresses. If the Pod is being used for mail storage for two domains, domain1 and domain2, then the Pod would have `/smail/domain1/` and `/smail/domain2/`. If domain1 has two users, user1@domain1 and user2@domain1, then the Pod would have `/smail/domain1/user1/` and `/smail/domain1/user2/`, each the equivalent of the "mailboxes" described in RFC 5322.

Inside of a mailbox there are, by default, two containers, one for messages (`.../messages/`) and one for directory indexes (`.../dir/`). A message is stored as a dataset by its `messageId` inside of the first (`.../messages/message1` for a message with a `messageId` of `message1`). For performance reasons discussed below in 5.2.2, Smail maintains indexes of each of the conceptual directories a message can be in (inbox, outbox, sent, drafts, and archive). These are stored as JSON files inside of the `dir` container (`.../dir/inbox.json` for the inbox directory). Each of these JSON objects contain an object with a single key, `contents`, which has a value of an object that itself contains objects (each key is the URL of a dataset for a message currently in the directory, and the value is an object of several values of the message, including subject and timestamp). The URLs of these JSON files are referenced by the `directory` property defined in the ontology. Smail offers no way to express such invariants, but a message's "directory" property should *always* refer to the directory file which contains a reference back to it (the message), and no other directory files should reference that message.

4.2 Server

The server application accomplishes two separate tasks: a sender reads messages from Smail's outbox and moves them to Smail's sent directory once they have been sent and a receiver listens for incoming emails, processes them, and saves them to Smail's inbox. These two components *can* be implemented separately, but for simplicity they have been combined into one application here.

The server application consists of two components, run in separate Docker containers and coordinated with Docker's Compose. The first is an off-the-shelf SMTP server with no code modification and little advanced configuration which is optimized to work in a containerized environment [1]. Including comments, this container is specified with 39 lines in the compose file.

The second component is the Smail-specific logic, which acts as a two-way relay between the Solid Pod and the SMTP server. This application is written in Node.JS, and requires 18 lines of specification in the compose file and an 8 line Dockerfile. An initial function starts both the receiver and sender services. Without comments, the receiver function requires 113 lines of code and the sender requires 126. Both rely on the ontology artifact described in section 4.1. Details of the development experience and performance are included in section 5.1.

4.3 Client

The presented client application is a React.JS application. React was chosen primarily because Inrupt provides a React SDK (`solid-ui-react`) for Solid, which is designed to work well with their general-purpose Solid JavaScript API library (`solid-client`) and authentication library (`solid-client-auth`) [3–5]. Excluding CSS styling code and some boilerplate JavaScript, the application is roughly 600 lines of code.

`solid-ui-react` provides components used for handling authentication, and `solid-client` provides logic for reading and updating Solid Pods. The ontology artifact described in section 4.1 is again used here.

5 Evaluation

Basic functionality (sending and receiving messages) has been achieved, demonstrating that Solid can indeed be used for mail storage. However, in the process of developing and testing Smail, we discovered a number of important caveats on that achievement that are potentially more important than the actual functionality of the model.

5.1 Development Experience

Given that Solid proposes a fairly substantial paradigm shift for web applications and services, it may be expected that developers accustomed to the incumbent model might find a bit of a learning curve. This was somewhat the case in our experience, but because the model itself is purposefully simple, it does not take long to figure out how to organize a system such as Smail at a high level.

At an implementation level, however, it became abundantly clear to us over the course of developing Smail that Solid,

both as a model and as an ecosystem, is immature. Documentation appears to be an especially weak point in the Solid Community. For example, the Solid Project Website does include a tutorial on creating ontologies, which provides a fairly comprehensive and understandable introduction to RDF and includes a step-by-step exercise. However, there are some contradictions between the body of the page and the example files it links to, and additionally there are several links which appear to be broken. Inrupt, the startup that has actually been developing JavaScript libraries that provide abstraction on top of HTTP requests to Pods, also has its own documentation. Generally, this is of higher quality than the guides published directly by the Solid Project, but is still lacking in examples. While poor documentation is not uncommon in software development, the lack of quality guidance in the Solid community seems particularly acute, and, in our opinion, exacerbates confusion inherent to a decentralized project led developed by a decentralized team.

5.2 Limitations

5.2.1 Lack of Reliable WebSocket Support

Though WebSockets are identified as a technology that Solid is built upon, this functionality appears to be largely unusable at present. Some Pod server software versions (including the one used to evaluate Smail) seem to simply not include support for the feature, and Inrupt's libraries appear to have alpha support, but we found this to be too unreliable to use in developing Smail. As a result, we use polling to check for outbound messages in `smail-server`.

5.2.2 SPARQL Support is Lacking

SPARQL Protocol and RDF Query Language (SPARQL) is a query language that is also a component technology of Solid that promises to provide an efficient and simple mechanism for querying linked data, to prevent the need for manual querying of individual Solid dataset and resultant processing and logic. Unfortunately, manual SPARQL querying of Solid Pods seems to be largely unsupported by current Pod server software, and implementation-level documentation is nonexistent.

As a result, in the case of Smail we found that we had to maintain indexes for each message directory. Were messages stored in a relational database, we would simply be able to issue a query such as `SELECT ID, SUBJECT, FROM, TIMESTAMP FROM MESSAGES WHERE DIR = "inbox";` in order to generate a list view for the inbox, and we believe SPARQL support would allow us to accomplish something similar with Solid storage. However, with only basic directory listing and file retrieval tools, we would have had to issue an HTTP request to determine the IDs of messages stored in a Pod, then issue an additional requests for each message to retrieve the other fields. In testing this approach with directories containing as few as 10 messages, we noticed message listing load times in excess of 2 seconds in `smail-client`, which we believed to be unacceptable.

Our solution, to maintain indexes as described above, improves performance, but it also betrays the simplicity envisioned by Solid and RDF by requiring application developers to concern themselves with non-RDF data. There are a number of operations that must be performed by both `smail-client` and `smail-server`, and the code for those operations is very similar between applications. In larger and more complex scenarios, it may make sense to create a library of functions specifically for interacting with certain ontologies that require similar multi-step updates. Given that users must inherently trust applications not to corrupt their data, it may be worthwhile for ontology developers to create such libraries even if SPARQL is more fully supported in the future.

Additionally, because updating messages in a Pod may require multiple HTTP requests (like in the case of index updates), concurrency concerns arise. These are left largely unaddressed in Smail, but the most direct approach to ensuring consistency may be to implement some sort of lockfile, which we believe would only discourage developers from attempting to build Solid-based applications.

5.3 Opportunities

5.3.1 Extensibility

Though Smail, as implemented, focusses on a basic set of features and does not address any of the "social" aspects of Solid, because multiple applications can write any ontology to a Pod, other applications could utilize the Smail ontology as a base and extend it with additional fields, or reference it from other, non-Smail datasets. As it exists now, Smail could support collaborative drafting of messages if users are willing to manage dataset access lists, but this could be further enhanced by other ontologies and applications specifically designed for this purpose.

5.3.2 Smail-as-a-Service

Smail, as presented, is a self-hosted application configured by a user. While this is a reasonable demonstration application, it is strongly recommended that such a setup be avoided for non-testing and non-development purposes. However, were a Smail-like effort continue to develop, webmail services could choose to offer Solid-based storage as an option. It is unlikely that services such as Gmail would be likely to take this path, but other, smaller and more specialized services may.

Acknowledgements

We would like to thank Professor Malte Schwarzkopf and Kinan Dak Albab for their advice and feedback on this project,

and for conducting such an interesting and rewarding course.

References

[1] docker-mailserver/docker-mailserver: Production-ready fullstack but simple mail server (SMTP, IMAP, LDAP, Antispam, Antivirus, etc.) running inside a container. https://github.com/docker-mailserver/docker-mailserver.

[2] Solid Applications. https://solidproject.org/apps.

[3] solid-client API — Inrupt solid-client API Documentation. https://docs.inrupt.com/developer-tools/api/javascript/solid-client/.

[4] solid-client-authn-browser API — Inrupt solid-client-authn-browser API Documentation. https://docs.inrupt.com/developer-tools/api/javascript/solid-client-authn-browser/.

[5] solid-client-authn-node API — Inrupt solid-client-authn-node API Documentation. https://docs.inrupt.com/developer-tools/api/javascript/solid-client-authn-node/.

[6] Solid Protocol. https://solidproject.org/TR/protocol.

[7] World's First Decentralized Email Solution | Blockchain based encrypted and free email service. https://ledgermail.io/.

[8] Artifact Generator. https://github.com/inrupt/artifact-generator, October 2021. original-date: 2019-07-03T11:33:44Z.

[9] Webmail. https://en.wikipedia.org/w/index.php?title=Webmail&oldid=1056155766, November 2021. Page Version ID: 1056155766.

[10] Jeff Gould. The Natural History of Gmail Data Mining. https://medium.com/@jeffgould/the-natural-history-of-gmail-data-mining-be115d196b10, November 2014.

[11] Pete Resnick. Internet Message Format. https://rfc-editor.org/rfc/rfc5322.txt, October 2008. Issue: 5322 Num Pages: 57 Series: Request for Comments Published: RFC 5322.

[12] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. Solid: A Platform for Decentralized Social Applications Based on Linked Data. http://cs.brown.edu/courses/csci2390/2021/readings/solid.pdf.

[13] Chris Sarli. smail-client. https://github.com/chris-sarli/smail-client.

[14] Chris Sarli. Smail ontology. https://chris-sarli.inrupt.net/voc/smail.ttl.

[15] Chris Sarli. smail-server. https://github.com/chris-sarli/smail-server.

[16] Ben Wolford. What is PGP encryption and how does it work? https://protonmail.com/blog/what-is-pgp-encryption/, August 2019.