PushPin: Towards Production-Quality Peer-to-Peer Collaboration

Peter van Hardenberg
pvh@inkandswitch.com
Ink & Switch, LLC
San Francisco, CA, USA

Martin Kleppmann
mk428@cl.cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Abstract

Fully peer-to-peer application software promises many benefits over cloud software, in particular, being able to function indefinitely without requiring servers. Research on distributed consistency mechanisms such as CRDTs has laid the foundation for P2P data synchronisation and collaboration. In this paper we report on our experience in taking these technologies beyond research prototypes, and working towards commercial-grade P2P collaboration software. We identify approaches that work well in our experience, such as the functional reactive programming paradigm, and highlight areas in need of further research, such as the reliability of NAT traversal and usability challenges.

CCS Concepts: • **Networks → Peer-to-peer protocols**; • **Software and its engineering → Peer-to-peer architectures**; *Synchronization*; • **Human-centered computing → Collaborative and social computing systems and tools**.

Keywords: real-time collaboration, CRDTs, peer-to-peer protocols, distributed programming, usability

ACM Reference Format:
Peter van Hardenberg and Martin Kleppmann. 2020. PushPin: Towards Production-Quality Peer-to-Peer Collaboration. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3380787.3393683

1 Introduction

In the past, software used to run on one computer. Today we expect our software and data to be available on multiple devices, with synchronisation across devices belonging to the same user (e.g. laptop, smartphone, tablet), and also allowing real-time collaboration between multiple users.

PaPoC '20, April 27, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-7524-5/20/04…$15.00
https://doi.org/10.1145/3380787.3393683

At present, such multi-user, multi-device software typically relies on cloud services that store the authoritative copy of the users' data, and which can be accessed through thin clients such as web browsers and mobile apps. However, reliance on the cloud comes with problems: services require ongoing maintenance by expensive 24/7 operations teams, and they cease to exist when the organisation backing them terminates their funding. If a cloud service shuts down, users lose access to all data stored in that service, unless some migration path to an alternative service is provided. Moreover, cloud-centric software often does not work well offline, and can be slow as the client waits for round-trips to the server in order to load or store data there.

Peer-to-peer (P2P) application software promises to overcome these problems. In principle, a P2P system should be able to function indefinitely, without depending on someone paying to keep a cloud service running. Storing an authoritative copy of data on the users' devices enables offline work and stronger data ownership [17], and synchronising updates through a P2P network should allow the same kind of real-time collaboration that we know from cloud software.

While various research prototypes of P2P collaboration software have been developed [25, 26, 37], we are yet to see any mainstream applications using this approach. The PushPin project, described in Section 3, is investigating if and how we can develop commercial-quality collaboration software with minimal reliance on servers. Our goal is not to create new algorithms or protocols, but rather to evaluate existing technologies by developing an example P2P application with a mindset of industrial software development best practices and realistic user requirements. We approach this project with broad interests in exploring programming models, P2P data distribution, reliability, and usability.

We build upon two foundational technologies:

Conflict-free Replicated Data Types (CRDTs), data structures that can be concurrently updated by multiple users on different devices, and which automatically synchronise and merge those updates [8, 33]. We discuss PushPin's use of CRDTs in Sections 3 and 4.

P2P replication protocols allow updates from one device to be propagated to other devices that have a copy of the data, without relying on cloud services [9, 28, 36]. We discuss these protocols in Section 5.

In summary, our findings are:

- CRDTs provide a reliable, principled foundation for P2P collaboration.
- Functional Reactive Programming (FRP) is effective for synchronising user interfaces with state managed by CRDTs (Section 4).
- P2P protocols work in many cases, but are poorly supported by many network routers (Section 5).
- Significant open problems remain, including around authentication and access control, indexing and search, schema evolution and compatibility, privacy, and the usability of systems where some devices are in sync while others are not.

2 Design Principles

Before going into the details of PushPin we outline the principles we applied to its design and development.

2.1 Local-First Software

While we want users' data to be accessible on multiple devices, authoritative copies of this data should reside on users' local computers. When servers are used, these replicas are primarily to facilitate data synchronisation and backup. In previous work we have coined the term *local-first software* to describe software that adheres to this principle [17].

With a local-first approach, the software continues working locally if the user's computer is disconnected from the Internet, and cross-device synchronisation happens in the background when a network connection is available. Even if all servers are shut down, the copy of the data on the local disk remains under the user's control, and fully available for both reads and writes.

2.2 Minimal Dependence on Servers

We wish for our software to avoid dependence on servers. Server infrastructure is fragile, and can be both expensive and laborious to operate and maintain. Further, it is obviously not available to clients without internet access. By architecting our software to avoid a server dependency, we improve resilience and durability of our software.

Thus, in addition to local data storage on each device, the cross-device data synchronisation mechanism should also depend on servers to the least degree possible, and servers should avoid taking unnecessary responsibilities. Where servers are used, we want them to be as simple, generic, and fungible as possible, so that one unavailable server can easily be replaced by another. Further, these servers should ideally not be centralised: any user or organisation should be able to provide servers to serve their needs.

2.3 Conflict-Free Data Synchronisation

The combination of using local on-device storage, support for offline editing, and synchronisation without servers implies that our application is *intrinsically distributed*. Each device serves as a replica, and it would not make sense to enforce any kind of single system image semantics across replicas, since that would imply that a device must wait for synchronous coordination with other devices whenever any data is changed. We cannot rely on consensus algorithms, which must wait for communication with a quorum of replicas.

Rather, we have to accept that each device has its own local view onto the shared data, and that those views may diverge as users update their data. As devices exchange updates, they converge again by merging their states. We do this using conflict-free replicated data types/CRDTs [33]; specifically, we use Automerge [13, 16], an operation-based CRDT implemented in JavaScript. This library provides fine-grained conflict resolution (see Section 3.2) for a rich set of datatypes, including character-by-character editable text [14] and JSON-like object trees [12]. PushPin requires this rich data model, which is not provided by other replicated data approaches such as Cloud Types [3].

This implementation model is not suitable for all types of software. For example, some corporate systems are built around robust hierarchies of access controls. While it may be possible in time to recreate these structures atop CRDTs, applications where data is primarily held by an individual are a more obvious first frontier to explore.

2.4 Mainstream, as Far as Possible

In order to explore the *user experience* implications of a peer-to-peer architecture, we wanted to develop not just a rough research prototype, but polished end-user software that is on par with commercial applications available today, with a thoughtful graphical and interaction design. We also wanted to explore the *developer experience* of peer-to-peer software, to understand how writing software with this architecture could become accessible to mainstream software engineers. Thus, we wanted to base our work on mainstream languages and platforms as far as possible.

3 PushPin: A Collaborative Corkboard

The PushPin software, shown in Figure 1, allows users to collect media of various types (including text, web pages, images, and PDF files), to archive and organise it. Media files are visually represented as *cards* on a two-dimensional *board*, where they can be resized and positioned arbitrarily. One board may be nested within another board, enabling hierarchical organisation and navigation. This board metaphor is inspired by software such as Hypercard [10], Miro [2] and Milanote [19]. PushPin is open source [4].

This application fits well with the principles articulated in Section 2: the data in this software belongs to the user, and there are essentially no restrictions as to what the user may do with their data. Unlike some systems (e.g. banking or payment systems, auction websites, ride-sharing services, or games), in this application there is no need to enforce any

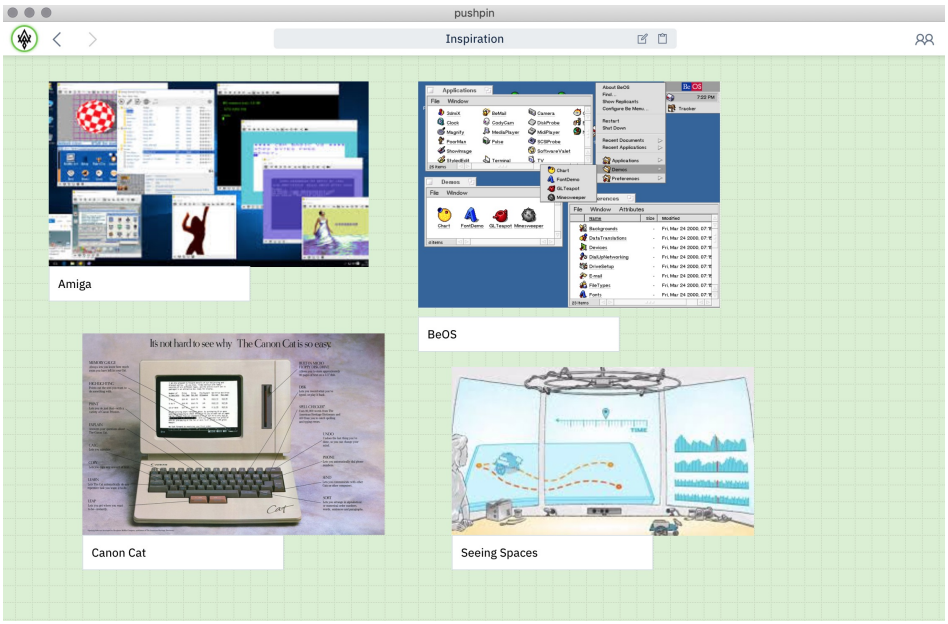

Figure 1. Screenshot of PushPin. The main user interface consists of cards of various types (text, image, PDF, ...) that can be freely arranged on a 2D "board". Boards can be nested within other boards. The toolbar at the top provides navigation between boards and sharing settings.

global rules or consensus across users, and there is no need for an authority to decide what actions are allowed. The only restrictions are access permissions (i.e. defining which user may view or modify which pieces of data), but we assume that a user with permission to edit some piece of data may modify it in any way.

3.1 Building Desktop Software with Electron

Web browsers have become the de-facto standard for cross-platform software development. They have a robust ecosystem of programming languages (e.g. TypeScript), frameworks (especially React [6]) and utility libraries (e.g. PDF rendering [24]). Browsers continue to become more powerful execution and rendering environments, with new features for HTML and CSS always evolving. They are also home to perhaps the largest community of software developers. Thus, by building PushPin within this ecosystem, we gain access to these tools, and hope to remain accessible to a large community of developers.

Unfortunately, web applications running in a traditional browser have constraints that make them unsuitable for local-first/P2P use. These constraints fall in two categories:

Storage. By default, browsers do not reliably store data or software. The Progressive Web Application standard [22] and APIs like localStorage or IndexedDB [21] provide some functionality to store local copies of code and data, but these systems are slow, and worse, often silently lose user data. These APIs are designed with the assumption that local user data is merely a cache, not authoritative, and such data is prone to being expired without warning or notification [34, 38]. As a result, users have no way of predicting whether their data or the application will be available offline.
Sometimes, manual steps are required: for example, Google Docs requires installing a special browser extension to enable offline support, and it only makes documents available offline when they are specifically selected by the user. Anecdotally, it appears to be common for users to think they had enabled offline usage, only to realise later that their data is unavailable.
Loss of user data is a cardinal sin, and the fact that browsers are unreliable keepers of locally persisted data is disqualifying for local-first implementations.

Networking. For security reasons, web browsers restrict the network communication that application code can perform: requests are restricted to client-server protocols such as HTTP or WebSocket, and are subject to the same-origin policy [23]. Even the built-in WebRTC peer-to-peer communication requires a centralised server to broker connections (see Section 5.1). It is not possible to use arbitrary TCP or UDP networking, which would be required to implement other peer-to-peer protocols.

We avoid these limitations by building on Electron [1]. Electron runs a JavaScript web application in a dedicated Chromium-based browser runtime, packaged as a downloadable and locally installed executable containing all of the

```
{
  "title": "Inspiration",
  "authors": [
    "hypermerge:/BvRmN2rU7pQBginzw9KqTXcUmtyYC5aCLiZZ314Ga8Vt?contentType=contact",
    "hypermerge:/6YTvUkNePGkPJH3fHaCoJfJdVdKbi6qjDKgp27u3SChG?contentType=contact"
  ],
  "cards": [
    { "x": 0, "y": 0, "width": 577, "height": 484,
      "url": "hypermerge:/4uhU1SDgy56cAo3dQH5tqSrrTAZnQSQKkEUPzNUXSvcV?contentType=image" },
    { "x": 0, "y": 486, "width": 577, "height": 95,
      "url": "hypermerge:/HHmxeCceWD1ZBrXKeHPv7k7umVW2ncsWyVrUiV3cmMYP?contentType=text" },
    { "x": 612, "y": 84, "width": 370, "height": 862,
      "url": "hypermerge:/E6qRRVUsbRcjddCV98TLyyRhPsVNgyiUXAdPsLyKM3mn?contentType=todo-list" }
  ]
}
```

Figure 2. JSON document representing a PushPin board with three cards on it.

required code. Once installed, the user can be sure that the application is available offline.

Electron also makes Node.js APIs available to application code, which enables full access to the local filesystem, and socket APIs allowing arbitrary TCP and UDP networking. Thus, Electron allows software engineers with web development skills to write cross-platform desktop software for macOS, Windows, and Linux.

3.2 Automerge Documents

Application state in PushPin is managed by Automerge [13, 16, 17], a JavaScript CRDT library that provides a JSON data model [12]. Automerge defines a format in which data updates can be written to local disk and replicated to other devices. Network communication is provided by a separate layer called Hypermerge, which we discuss in Section 5.

An Automerge document is the unit of replication and sharing in PushPin: that is, a user can access either all or none of a document. We want a PushPin user to be able to share their content with other users at a fine-grained level: e.g. one card at a time, or one board at a time. For this reason, we represent each card and each board as a separate document.

For example, Figure 2 shows the JSON representation of a board. It contains a `title` (rendered in the toolbar in Figure 1), a list of users who have contributed to the board, and a list of cards on the board. Each card has `x`, `y`, `width` and `height` attributes recording its dimensions, and a `url` linking to the document that stores its content (see Section 3.3).

Automerge allows state to be concurrently updated on different devices, and ensures that replicas converge to the same state as they communicate. For example, if a user drags a card to a new position, this modifies only the `x` and `y` attributes of that card, leaving the rest of the document unchanged. If another user concurrently creates a new card on another device, it is inserted into the list of cards. The CRDT tracks that change and records it as an Automerge operation for replication. The concurrent updates (changing the position of one card and adding another card) are merged cleanly on each replica.

3.3 URLs and Linking

Each document is identified by a unique `hypermerge` URL of the form shown in Figure 2. Given a URL, a PushPin instance can obtain the corresponding document content through a process described in Section 5.2. By including one document's URL in the content of another document, we form a graph of links, similar to the web.

The same URL can be referenced from multiple places, allowing e.g. the same card to be embedded on multiple boards. URLs can also be shared with another user by sending them through any communication channel, such as email. When a PushPin instance loads a document containing a `hypermerge` URL, it eagerly resolves and downloads the content belonging to that URL. Thus, any transitively reachable documents are automatically added to PushPin's local document storage on that device, making them available offline.

Each URL also includes a `contentType` parameter, which indicates how that document should be rendered in the user interface. This parameter is part of the URL, not the document content, because the same document content may be rendered differently in different contexts. For example, PushPin could be extended to support flashcards for language learning. In one context, the document containing the database of flashcards could be rendered as a list of entries, while in another context it might be rendered as a quiz interface, presenting one side of one flashcard at a time.

4 Creating User Interfaces for CRDTs

In traditional server-centric web applications, the server is considered the sole authority, and synchronisation between

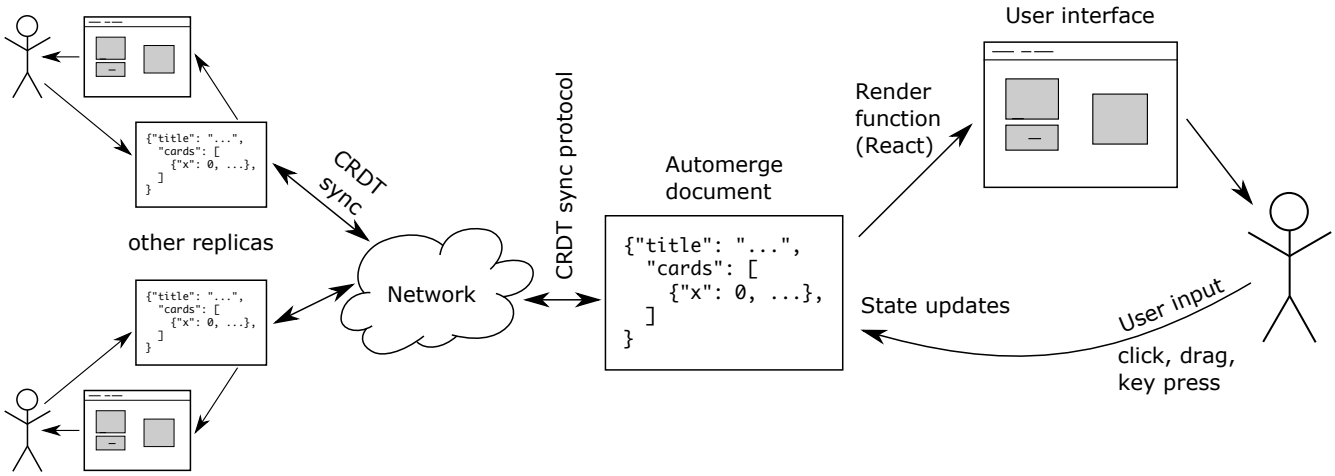

Figure 3. The data flow of Document Functional Reactive Programming (DFRP).

client-side and server-side state is usually performed in an ad-hoc way, with user actions translating into HTTP requests that perform API calls on a server. Handling the responses to those HTTP requests typically leads to "callback hell" and requires lots of tricky application-specific error handling code [20].

In contrast, CRDTs provide us with a principled way of managing and reasoning about the state of multiple replicas: each replica can optimistically update its local state, and a replication protocol running in the background ensures that they eventually converge [32]. Application code only interacts with the local CRDT data; no application-specific code is required to handle errors such as network timeouts.

However, the convergence we have discussed so far is at the level of Automerge documents, such as the JSON shown in Figure 2. In this section we will expand the discussion to include the state of the user interface.

4.1 Functional Reactive Programming (FRP)

For PushPin we wanted a similarly principled approach to developing user interfaces, which would allow the state of the CRDT and the state of the user interface to be kept in sync in a way that is robust and easy to reason about.

We solved this problem using the Functional Reactive Programming (FRP) approach [5, 20], which has been gaining popularity in JavaScript web applications, especially due to its implementation in Facebook's React library [6].

FRP works by defining a deterministic *render* function that takes the current state of an application and returns a description of a user interface that displays this state. Whenever the user performs an action, rather than updating the user interface directly, the *state* is updated to reflect that action. The updated state is then passed to the render function, which computes the updated user interface. Re-computing the full user interface on every change would be expensive, so in practice, performance is improved by detecting which elements of the view rely on changed state, and updating only those parts that are necessary. However, the conceptual simplicity remains: the render function cleanly translates between application state and user interface state.

4.2 Document FRP (DFRP)

In the usual implementation of FRP, the application state (the input to the render function) is simply an in-memory object. In PushPin, we generalise this approach by making an Automerge document the input to the render function. We call this approach *Document FRP* (DFRP).

Figure 3 illustrates the DFRP data flow. An Automerge document may be newly created, or loaded from local disk, or fetched over the network. Regardless of its origin, the document can be displayed by passing it to the deterministic render function.

The resulting user interface has attached event handlers that detect user input, such as mouse clicks or keyboard input. When any such events occur, the Automerge document is updated to reflect the user input, and the render function is called again to refresh the user interface. This local interaction can be performed regardless of whether the user is currently connected to the Internet.

Likewise, whenever the network synchronisation protocol running in the background receives an update from another replica, Automerge applies this update to the CRDT state. We then again call the render function with this updated state, which refreshes the user interface accordingly. Thus, the user interface logic need make no distinction between a local user's updates and remote updates received over the network: both can simply result in a call to the render function. (It may be desirable to differentiate writes from different sources, but PushPin does not do so in the current implementation.)

In applications written in the DFRP style, the developer never needs to worry about API calls to a server backend, or

the fact that they may fail. Instead, all communication is via shared CRDT state that can be updated by any collaborator at any time. Because a DFRP project's consistent data model doesn't require implementing and integrating separate codebases for clients and servers, a DFRP project can actually be *simpler* to develop than traditional web applications, even though it is also more powerful.

4.3 Ephemeral Versus Persistent State

The most rudimentary approach to DFRP is to keep all application state in a single CRDT. This is problematic: for some types of state, such as cursor or page scroll position, each PushPin instance should have its own state — converging all replicas to the same state is undesirable.

Also, some fast-changing data (e.g. mouse position or animation timers) could swamp the CRDT with low-value information; there is no reason to persist such updates. We therefore divide the application state into two parts: persistent state that is replicated, and ephemeral state that exists only locally.

Still, we may wish to share ephemeral data with peers that are currently online: it can be helpful to see whether other users are viewing the same document as you, where their cursor is located, or which items they have selected. It might be interesting to see another user's current position in a podcast or video, or a hint that shows they are currently typing into a chat textbox but have not yet submitted.

For ephemeral updates PushPin uses an additional messaging channel, adjacent to the CRDT, which ties arbitrary messages to a device and user context. The current implementation is rudimentary: ephemeral data is not associated with a particular CRDT state and is distributed only over direct P2P connections. Nevertheless, it enables shared contextual awareness in the user experience of PushPin, providing a feeling of presence when other users are online or collaborating.

4.4 Supporting Multiple Documents

Sections 3.2 and 3.3 explained how PushPin breaks down all of its data into fine-grained documents. This decomposition of state is also beneficial for user interface development, because rendering can be done at the document level.

Monolithic FRP can become difficult to reason about, as all events and states are unified into a single transition function. In contrast, PushPin's fundamental primitive is to render one Automerge document (e.g. the contents of one card) with one FRP function. These document renderers can be nested arbitrarily to produce complex views.

For example, the board document *render* function can include renderings of text or image documents within its output. Likewise, the same FRP code could render a text document at the root of the application to show the content full-screen. This makes the user interface components more easily reusable and composable. It is also easy to extend the application with new renderers, allowing users to have boards containing a mixture of predefined card types and custom card types they have defined themselves.

Moreover, we can provide multiple renderers that are able to render the same document in different ways. For example, in the user interface shown in Figure 1, the board is actually being rendered twice: firstly as a spatial representation containing the cards, and secondly within the toolbar as the title of the board and its authors. Some *render* functions can even render a wide variety of documents: for example, the toolbar can render any type of document that has a title and a list of authors.

4.5 High-Performance User Interfaces

To deliver high-performance software, we need to respond as fast as the user's display. For a monitor with a 60 Hz refresh rate, the application should respond to input within 16 ms.

Unfortunately, many tasks including CRDT computations and cryptographic operations can easily exceed this time budget, and so we perform as much work as possible in a background process. Automerge is split into a frontend that runs on the user interface thread, and a backend that runs on the background thread, with an asynchronous message-passing protocol between the two.

Moreover, Automerge allows local user input to be applied immediately to the document state in the frontend, without waiting for any communication with the backend. This minimises the time between user input (such as pressing a key) and the corresponding update appearing on the screen. Automerge transparently handles the fact that local user input in the frontend can happen concurrently with the backend processing remote operations. We have not seen any other CRDT implementation that provides this feature.

5 Peer-to-peer Networking for Collaboration

Every networked application relies on three core facilities provided by the networking stack: discovering the network address to connect to, establishing a connection, and securing the confidentiality and integrity of the data transfer.

In a traditional web application, discovery is provided by DNS, connection by TCP, and security by SSL/TLS. However, these technologies are designed for centralised infrastructure, and they are not a good fit with our goal of being resilient to infrastructure failures:

- DNS records are only available with internet access, and expire if the domain name owner stops paying the required registration fees.
- TCP requires a server with a publicly routeable IP address; it cannot connect directly to most end-user devices as they are behind NAT (see Section 5.3).

- SSL/TLS certificates are tied to DNS records, which mean only servers associated with that certificate can be trusted to provide data.

The peer-to-peer technologies we explored in PushPin attempt to overcome the need for centralised infrastructure.

5.1 Existing Peer-to-Peer Technologies

We considered several P2P networking stacks for PushPin:

WebRTC is a peer-to-peer protocol built into modern web browsers. It is primarily designed for audio and video calls, but it can also carry application data. WebRTC does not provide a peer discovery mechanism; typically, applications rely on a server to help peers discover each others' IP addresses (this process is called *signaling*).

BitTorrent is widely used for file sharing. It provides a distributed hash table (DHT) for peer discovery, and uses the uTP protocol to establish connections between peers. However, it is designed for static files, and is not suitable for replicating data that is constantly changing, like in collaboration software.

IPFS aims to provide decentralised storage through a networking stack called *libp2p*. Like BitTorrent, it is mostly focused on replicating static files; it provides limited support for changing data through its IPNS and PubSub modules, but these features are immature at the time of writing.

Dat [11, 28] is a peer-to-peer data sharing platform. For peer discovery it uses a combination of mDNS on local networks (see Section 5.2), and a DHT with seed nodes operated by a nonprofit foundation. It uses BitTorrent's uTP protocol to establish connections.

For PushPin we decided to build upon the *hypercore* protocol and implementation from the Dat project [11, 28]. A hypercore is an append-only log that is authenticated with a public key; only the owner of the corresponding private key can modify the log, but many peers can store replicas of it.

We map each Automerge document to a set of hypercores, with one hypercore per device that has edited the document. We chose this protocol since it focuses on replicating mutable data, unlike e.g. WebRTC, which provides only an ephemeral messaging channel. Our integration of Hypercore with Automerge is called *Hypermerge* [29].

5.2 Peer Discovery

The Dat protocol allows the replicas of a hypercore to be discovered based on a hash of its public key [11]. We construct the URL for a document by encoding this public key. Thus, the URL is a stable identifier for the document, even as its content changes, and knowledge of the URL allows a peer to obtain a copy of the document via the peer discovery mechanism.

When peers are on the same LAN (wired or wireless network), they attempt to discover one another using mDNS, a variation on DNS that broadcasts DNS-like service advertisements or discovery requests to a well known multicast IP address [35]. (mDNS is also known as *Bonjour* or *Zeroconf*, and is commonly used for discovering local network resources such as printers.) If successful, the peers can connect directly via TCP and begin exchanging data. This mode of discovery is appealing since it depends only on the local network: communication between peers does not flow via the Internet, and it does not depend on any centralised infrastructure.

When peers are not on the same LAN, the Dat protocol queries a Distributed Hash Table (DHT) for peer discovery [30]. A DHT operates by discovering new peers during its operation, but for bootstrapping purposes the addresses of a few seed nodes have to be hard-coded into the client software. In PushPin's case, the seed nodes are operated by the Dat non-profit foundation (the Dat DHT is independent from the BitTorrent DHT). This is a weak form of centralisation: if all of those nodes become unavailable, a software update would be necessary, but in principle there is no reason why those nodes could not be distributed more resiliently across multiple organisations.

5.3 NAT Traversal

Due to a shortage of IPv4 addresses, most personal computing devices do not have a globally reachable IP address, but rather a local address in a reserved space (e.g. 192.168.x.x or 10.x.x.x). When such a device wishes to establish a connection to another, the local router records the destination of outbound traffic and routes responses back to the originating local client. This process is called Network Address Translation (NAT).

A device behind a NAT router can make outbound connections, but it cannot receive inbound TCP connections from outside of the NAT. An exception: in home environments, where the user has control over their own router, the UPnP standard allows devices to reserve particular ports on the router's public IP address as the destination for inbound connections. However, mobile devices often use networks with NAT on which UPnP is not available, such as a coffee shop WiFi or a corporate office network.

In these cases, the most viable solution is known as "hole punching" or NAT traversal [7]. This process requires the temporary intervention of a third host to introduce the two peers, and both peers sending UDP packets to each other, allowing a connection to be established. NAT traversal is performed by BitTorrent's uTP protocol [27], and by the STUN protocol in WebRTC [31].

However, there are situations in which neither the LAN discovery approach nor NAT traversal works. For example, some coffee-shop WiFi and some corporate networks are set up in a "guest network" mode, which prevents all local

connections between devices on the network (intended as a security measure to prevent inadvertent sharing of data with other users on a public network). Without local traffic, we attempt to fall back on NAT traversal; however, this approach also fails, since many routers in their default configuration refuse to create NAT traversing routes that originate and terminate within the same network.

In this case, establishing a direct connection between the peers seems to be impossible, and the only remaining option is to use a server with a public IP address to proxy the communication between the peers, e.g. using the TURN protocol [18].

In the case of PushPin, users in this position can take advantage of indirect replication through another mutually routable peer. In fact, this is one of the key benefits of running a storage peer described in the next section. Note that, while we have not implemented it today, in principle, any PushPin instance could provide relay services to other peers.

5.4 Storage Peers

A limitation of any peer-to-peer system is that two peers can only communicate while they are both online. However, mobile devices are often offline, potentially making it difficult to find an opportunity to exchange updates. For example, if your colleague has shared the URL of a PushPin board with you, it would be annoying if you could not access that board because the only copy of the board resides on your colleague's closed laptop.

We can overcome this limitation by introducing *storage peers*, which replicate all the data belonging to a particular user or set of users. A storage peer runs the same replication code as PushPin, recursively following any Hypermerge URLs, but it runs as a simple Unix daemon without any visual user interface. The storage peer can be deployed on a server or other computer that is always online, allowing other devices to sync with the storage peer at any time. Storage peers also provide a form of backup.

Unlike a traditional server, a storage peer can be reached through NAT traversal, so it does not need a public IP address: for example, it could be a device on the user's home internet connection. Since it only stores the data for a small number of users, it does not need to be a powerful machine. We have experimented with using a Raspberry Pi as storage peer, writing data to an SD card, and also running storage peers on virtual server instances in public cloud providers.

An interesting detail of the PushPin storage peer is that its user interface is not a web interface, nor a command line interface, but an Automerge document. When it starts up, the storage peer prints a URL that users can load from PushPin on another computer. Users then interact with their storage peer by making edits to this administration document within PushPin. The storage peer daemon monitors that document for changes and takes actions based on what it finds.

5.5 Data Confidentiality and Integrity

Some peer-to-peer systems, such as BitTorrent and IPFS, use a content-addressable storage approach to data integrity: files are identified by the hash of their contents, and the recipient can check the integrity of the file by comparing its actual hash to the expected hash. However, this approach is not suitable for collaboration software, where the shared data changes frequently, and the hash would change on every modification. Instead, Dat relies on cryptographic signatures to ensure that nobody can make undetected alterations to the data without knowing the private key.

The Dat protocol encrypts the communication between peers, using the URL of the document as a shared secret to establish the encryption key [11]. Thus, the URL acts as a bearer token (or capability) that grants read access to the document to anyone who knows it.

A downside of this approach is that there is no real way of revoking a user's access to a document. It is possible to generate a new URL and copy the document content, but this breaks any links to that document. We are interested in stronger access control and end-to-end encryption protocols for collaboration software that allow key rotation and revocation [15], but we have left this issue out of scope in the PushPin project.

The peer discovery protocol uses hashes of URLs, not the URLs themselves, so that anyone observing the peer discovery traffic (such as the DNS server, or other devices on the local network when using mDNS) does not gain the ability to read the document.

5.6 Scalability of Peer Discovery

On a local network, a Dat peer broadcasts all the hashes of URLs for data it holds, and all the hashes it requests. This approach is suitable when the number of URLs is small, but it breaks down as collections expand. Since PushPin uses a separate URL for each card, a user quickly accumulates many hundreds or thousands of document URLs. In our testing we found that we could fairly reliably crash the consumer-grade WiFi routers found in short-term rentals with half a dozen researchers sharing their collections.

The Secure Scuttlebutt [36] project avoids this problem by placing all of a user's activity into a single log, which it then merges with all of that user's peers and their peers out for several degrees of social connection. This trades one problem for another: by merging all of a user's data (and their peers' data) into one feed, there is no way to selectively synchronise that user. A peer either downloads all, or none.

Fortunately, we have observed that users tend to collaborate on more than one document with the same collaborator. Thus, when searching for peers that have a copy of a new document, it is likely that this document of interest can be found in the repository of a peer you are already connected to. We can significantly reduce the amount of discovery network

traffic by first querying existing peers, and only performing a global DHT lookup if this fails. Perhaps these peers could also forward queries on our behalf, recreating a DHT-like network. This is an area for future work.

5.7 Metadata Privacy

A downside of peer-to-peer protocols is that the peer discovery mechanisms leak information about users to other nodes on the network. Although the content of documents is only available to peers who know their URLs, the discovery keys (hashes of URLs) are widely broadcast, allowing a user's device to be identified by the pattern of discovery keys it shares. With this information, an attacker can monitor a user's IP addresses over time, and thus track their approximate physical location.

A number of defences could reduce this tracking potential (e.g. automatic rotation of discovery keys, or some form of interactive proof exchange through a third party prior to exposing IP addresses), but this remains an area of active concern and research.

6 Conclusions

The PushPin project set out to create a novel type of software: fully-featured peer-to-peer collaboration software that is fast, reliable, and useful. Specifically, we wanted the software to:

- respond quickly (within 16 ms) to local user input;
- permit local reads and writes at all times;
- allow collaboration over any kind of network connection (with or without internet access);
- not rely on any centralised services.

By storing data in the Automerge CRDT, replicating it over a peer-to-peer network based on Dat, building user interfaces using React's FRP model, and delivering the software as an Electron application, PushPin has met these goals.

6.1 State Synchronisation

Taking a principled approach to state synchronisation reduces the complexity of developing distributed software. We synchronise replicas with the Automerge CRDT, and we use FRP to synchronise the user interface with the local application state. This programming model treats all state updates equally, regardless of origin, eliminating ad-hoc calls to remote APIs. CRDT updates can be replicated immediately for real-time collaboration, or batched up and sent asynchronously. We achieve high performance by splitting Automerge into a frontend and backend, running expensive operations on a background thread without blocking the user interface.

Complex application states are composed of several related documents (e.g. directories, text notes, user profiles), each with their own context-specific rendering function. The URLs for these documents are self-validating and stable over time.

6.2 Networking

Peer-to-peer networking has three stages: content discovery, connection, and synchronisation. The most promising techniques for discovery are based on the distributed hash tables, supplemented by mDNS locally. Connection is complicated by the widespread assumption of client-server architectures and widespread NAT routing, and connectivity in public environments like cafes and corporate offices can be particularly challenging. The state of the art for NAT traversal in public environments is "hole punching", which PushPin performs using uTP, a NAT-traversing protocol inherited from BitTorrent, but it is imperfect. As with BitTorrent, by replicating self-validating data, the system avoids many concerns about trusting data origin.

6.3 Future Work

Many open problems remain, and we invite the community to help explore these topics in future work:

- Different versions of an application, running different data model versions, need to be able to run side-by-side and interoperate. How do we enable this, while preventing documents from entering invalid states?
- Establishing direct peer-to-peer connectivity is problematic in certain common network environments.
- Distributed Hash Table side channels are a concerning source of privacy leaks.
- The PushPin implementation currently has no mechanism for determining which users' writes to a document should be accepted.
- In a P2P system, different peers may have seen different subsets of updates for a given document. How do we communicate this to users?
- New and old networking stacks have promise to improve connectivity, including BLE, WiFiDirect, and ultrasonic modems.
- PushPin currently leaves almost all data un-encrypted, requiring trust in any peer that stores your data.
- Rotation of cryptographic keys is an important issue that we are exploring.
- Efficient document querying and indexes are needed: there is no way to load just the titles of a collection of Automerge documents.

Acknowledgments

Thank you to Roshan Choxi, Ignatius Gilfedder, Mark McGranaghan, Jeff Peterson, and Matt Tognetti, who contributed to the development of PushPin. Thank you to the Dat Foundation, particularly Mathias Buus, for their development of dat, hypercore, and hyperswarm. The project was produced under the auspices of the Ink & Switch research lab (https://www.inkandswitch.com/). Martin Kleppmann is supported by a Leverhulme Trust Early Career Fellowship and by the Isaac Newton Trust.

References

[1] [n.d.]. Electron. https://www.electronjs.org/

[2] [n.d.]. Miro. https://miro.com

[3] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. 2012. Cloud Types for Eventual Consistency. In *26th European Conference on Object-Oriented Programming (ECOOP 2012)*. Springer LNCS, volume 7313, 283–307. https://doi.org/10.1007/978-3-642-31057-7_14

[4] Roshan Choxi, Ignatius Gilfedder, Mark McGranaghan, Jeff Peterson, Matt Tognetti, and Peter van Hardenberg. 2019. PushPin source code. https://github.com/automerge/pushpin

[5] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *34th Annual SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. ACM, 411–422. https://doi.org/10.1145/2491956.2462161

[6] Facebook, Inc. [n.d.]. React. https://reactjs.org/

[7] Bryan Ford, Pyda Srisuresh, and Dan Kegel. 2005. Peer-to-Peer Communication Across Network Address Translators. In *USENIX Annual Technical Conference (ATC 2005)*. 179–192. http://brynosaurus.com/pub/net/p2pnat.pdf

[8] Victor B F Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, OOPSLA (Oct. 2017). https://doi.org/10.1145/3133933

[9] Richard Guy, Peter Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. 1999. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *International Conference on Conceptual Modeling Workshops*. Springer LNCS, volume 1552, 254–265. https://doi.org/10.1007/978-3-540-49121-7_22

[10] Internet Archive. [n.d.]. HyperCard Stacks. https://archive.org/details/hypercardstacks

[11] Duncan Keall. 2019. How Dat works. https://datprotocol.github.io/how-dat-works/

[12] Martin Kleppmann and Alastair R Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (April 2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

[13] Martin Kleppmann and Alastair R. Beresford. 2018. Automerge: Real-time data sync between edge devices. In *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018)*. https://mobiuk.org/abstract/S4-P5-Kleppmann-Automerge.pdf

[14] Martin Kleppmann, Victor B F Gomes, Dominic P Mulligan, and Alastair R Beresford. 2019. Interleaving anomalies in collaborative text editors. In *6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2019)*. ACM. https://doi.org/10.1145/3301419.3323972

[15] Martin Kleppmann, Stephan A Kollmann, Diana A Vasile, and Alastair R Beresford. 2018. From Secure Messaging to Secure Collaboration. In *26th International Workshop on Security Protocols (SPW 2018)*. Springer LNCS, volume 11286, 179–185. https://doi.org/10.1007/978-3-030-03251-7_21

[16] Martin Kleppmann, Peter van Hardenberg, Orion Henry, and Herb Caudill. [n.d.]. Automerge. https://github.com/automerge/automerge

[17] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You own your data, in spite of the cloud. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. ACM, 154–178. https://doi.org/10.1145/3359591.3359737

[18] R. Mahy, P. Matthews, and J. Rosenberg. 2010. RFC5766: Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). https://doi.org/10.17487/rfc5766

[19] Milanote Pty Ltd. [n.d.]. Milanote. https://www.milanote.com

[20] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, Vol. 109. Schloss Dagstuhl. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1

[21] Mozilla Developer Network. [n.d.]. IndexedDB API. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

[22] Mozilla Developer Network. [n.d.]. Progressive web apps (PWAs). https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

[23] Mozilla Developer Network. [n.d.]. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[24] Mozilla Labs. [n.d.]. PDF.js. https://mozilla.github.io/pdf.js/

[25] Brice Nédelec, Pascal Molli, and Achour Mostefaoui. 2016. CRATE: Writing Stories Together with our Browsers. In *25th International World Wide Web Conference*. ACM, 231–234. https://doi.org/10.1145/2872518.2890539

[26] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *19th International Conference on Supporting Group Work (GROUP 2016)*. ACM, 39–49. https://doi.org/10.1145/2957276.2957310

[27] Arvid Norberg. 2009. BEP 29: uTorrent transport protocol. https://www.bittorrent.org/beps/bep_0029.html

[28] Maxwell Ogden, Karissa McKelvey, and Mathias Buus Madsen. 2018. Dat – Distributed Dataset Synchronization and Versioning. https://github.com/datprotocol/whitepaper/raw/master/dat-paper.pdf

[29] Jeff Peterson, Peter Hardenberg, Matt Tognetti, Jim Pick, and Orion Henry. [n.d.]. Hypermerge. https://github.com/automerge/hypermerge

[30] RangerMauve. [n.d.]. How Dat discovers peers. https://rangermauve.hashbase.io/posts/how-dat-discovers-peers

[31] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. 2008. RFC5389: Session Traversal Utilities for NAT (STUN). https://doi.org/10.17487/rfc5389

[32] Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *Comput. Surveys* 37, 1 (March 2005), 42–81. https://doi.org/10.1145/1057977.1057980

[33] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*. Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[34] Stack Overflow. [n.d.]. When is localStorage cleared? https://stackoverflow.com/questions/8537112/when-is-localstorage-cleared

[35] Daniel H Steinberg and Stuart Cheshire. 2005. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media.

[36] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. 2019. Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications. In *6th ACM Conference on Information-Centric Networking (ICN 2019)*. ACM. https://doi.org/10.1145/3357150.3357396

[37] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *26th International Conference on World Wide Web (WWW 2017)*. ACM, 283–292. https://doi.org/10.1145/3038912.3052673

[38] John Wilander. 2020. Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/