Persistent Streams: The Internet With Ephemeral Storage

Oskar Carl, Peter Zdankin, Matthias Schaffeld, Viktor Matkovic,
Yang Yu, Timo Elbers, Torben Weis
Distributed Systems, University of Duisburg-Essen
Duisburg, Germany
{firstname}.{lastname}@uni-due.de

ABSTRACT

Communication in the internet today is transient by default. Because of this, whenever an application needs to store data even for a moment, its provider needs to develop an application-specific solution, which is often done using client-server models. This is costly at scale, and generally requires users to concede control over the data they generate to allow application providers to generate revenue from the collected data to finance the operation of these servers. This also leads to a lock-in effect, which is prohibitive for new applications entering a market. To solve these issues, we propose *persistent streams*, an application-agnostic communication protocol that includes ephemeral and persistent storage and is able to handle both discrete as well as continuous (streaming) data. Including storage into the communication path removes the need for application servers completely. Even though the protocol relies on (cloud) servers as transmission and storage proxies, we expect the emergence of new storage technologies like non-volatile main memory to alleviate some issues this introduces. We also show the general applicability of this solution using different kinds of applications as examples. Overall, persistent streams have the potential to greatly reduce the burdens on application providers while also enabling users to exercise increased control over their data.

KEYWORDS

communication protocols, decentralization, interoperability, federation, non-volatile memory

1 INTRODUCTION

Providing an application that communicates over the internet today requires additional infrastructure to exchange data between its users. Popular solutions include building upon an existing network or application, hosting dedicated servers, or using peer-to-peer (P2P) networks. Each of these is associated with different kinds of drawbacks and expenses. In the case of common client-server solutions, the critical issue for users is a lock-in effect, in which certain services can only be used as a user of its associated platform or application. Interoperability between services needs to be built explicitly and relies on the platform owner either creating it or allowing access to its data in a way that allows external extensions. This also commonly prohibits users from choosing their interfaces for communication based on personal preferences or exercising control over their generated data.

Thus, even though the internet was built based on decentralized ideals, today it is often not practically possible for users to choose their preferred applications for communication or even decide which third party has access to their data. With E-Mail it is simple to select a provider. It is possible to set up a server personally on arbitrary infrastructure, and a multitude of front-end applications are available to send and receive mail. This is independent of which provider and application the other parties in a (group) communication are using. The same is not possible for most modern communication platforms, which provide central servers and specific applications that must be used exclusively in order to reach people on their network.

This paper discusses current approaches to handle communication and data storage in distributed platforms in Section 2 and depicts prominent use cases and issues when implementing solutions for these using current approaches in Section 3. Then we describe a novel approach to amend these issues in Section 4. Finally, general ideas for efficient handling of data storage for reliable transmission are discussed in Section 5.

2 STATE OF THE ART

Currently, most architectures in large distributed platforms are client-server based. In this paradigm the users obtain the application from the provider and also send all communication to the provider's servers, which distribute them. Figure 1 shows a simple workflow for a chat application being used by two users. If user A wants to send a message to user B

FGBS '21, September 21–22, 2021, Trondheim, Norway

https://doi.org/10.18420/fgbs2021h-01

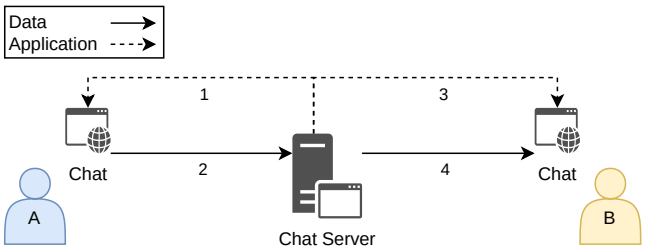

Figure 1: Abstract depiction of *sending a message* in a common client-server oriented model.

using the browser-based chat application, they access the application (1) hosted on the provider's server, and send their message (2) back to the server. The server then caches this message until user B accesses the application (3) and retrieves the message (4). This displays how users are required to relinquish control over their data and are completely reliant on the provider for both availability and data protection, since all steps of the process rely on the provider's server being accessible and able to fulfill their requests, making each one a single point of failure in its respective application.

In recent years it has become evident that user data is not properly protected in many cases [2] and even large providers such as Zoom[1] and Twitter[2], – amongst others – occasionally suffer from availability degradation and failures. If one such centralized service suffers from a global outage, all of its users are cut off from using the applications. Even when outages are regionally isolated, large numbers of users are unable to utilize the services and access their data.

A number of approaches to remove the strong coupling between data and applications to solve common privacy issues in client-server approaches have been created. However, none of them is able to provide means of communication for general application across a wide range of prominent use cases, which include both discrete as well as continuous (streaming) data.

ActivityPub [9] is one attempt at solving the issue of platform lock-in of users. It does this by allowing multiple *instances* of a platform to *federate* content created by users freely among them. However, data generated by a user still lies in the custody of the instance administrators, requiring a certain level of trust between them and their users. Additionally, since its internal data structure, ActivityStreams [7], is based on JSON-Linked Data, transferring arbitrary binary data can be inefficient and open-ended stream transmission is impossible.

[1]TechHQ on the Zoom outage on Aug. 23rd, 2021: https://techhq.com/2021/08/zoom-outage-due-to-overreliance-or-videoconferencing-fatigue/

[2]The Guardian on the global Twitter outage Oct. 16th, 2020: https://www.theguardian.com/world/2020/oct/16/twitter-outage-social-media-platform-goes-down-across-the-world

The Solid platform [4] aims to remove the strong coupling between the application provider and user data storage. Applications are required to request authorization to use specific parts of the user's *personal online datastore* (pod) stored independently at another provider. It is built with social web applications in mind, but since it is able to handle arbitrary data types, it can also be used in other contexts. However, being based on RESTful APIs, it cannot be used for arbitrary types of communication like streaming and similar push-based contexts.

The *InterPlanetary File System* (IPFS) [1] is a distributed file system based on a combination of well-known P2P and version control technologies. Its aim is to create a single global namespace of content distributed among peers all over the world. This creates a few issues, since there needs to be an incentive for hosters to provide peers capable of handling potentially large amounts of storage and requests. Since it is based on P2P technologies, it also experiences issues like blocking from firewalls. Since files are internally represented using immutable objects, streaming is not possible.

Live collaboration of multiple users is another central issue. While it is possible to solve this in federated networks, as shown using operational transformation in Google Wave [10], the Wave protocol only supports static contents with atomic modifications. Again, streaming of data is not supported. Similarly, synchronization of actions in P2P networks has proven to be a complex task. Results are usually also imperfect, since a solution always requires concessions in at least one of *partition tolerance*, *availability*, or *strong consistency*, and also needs to be able to handle long latencies between peers [6]. On the issue of private storage of data in public clouds, research has shown that it is possible in specific contexts [8]. However, the solution requires extensive setup and resources on the user side, since encryption and decryption need to happen outside of the cloud providers' control. It may also cause issues in performance and concurrent file access and does not support streaming of open-ended data.

All of this previous research has led us to believe that a general solution for private, user-controlled communication needs to be a) a lightweight *protocol*, only handling binary data without making assumptions on its structure, b) delegate the task of integrating layers above transport encryption to the applications, and c) support synchronization and live streaming of data in addition to persistent storage.

3 DISTRIBUTED APPLICATIONS

To evaluate whether an approach for a general communication protocol is usable in everyday applications, we first need to define use cases. While the current landscape of the internet includes many different kinds of applications, this section depicts only a few of them to display limitations

and deficiencies of current solutions. It also contrasts popular client-server solutions with the distributed approaches presented in Section 2.

The first application is a microblogging platform, where users can post text and other static content for arbitrary audiences to consume. Twitter solves this in a common client-server model, where all data is stored on the application servers and visibility is enforced by them. To give users more control over their data, Mastodon is an ActivityPub-based application, that was created as a replacement for Twitter using federation. However, since federation utilizes instances to distribute the content, it is not possible in Mastodon to define arbitrary audiences without giving all instances of users in the audience access to the content.

The second use case is public live streaming of video content. As an example of a common client-server implementation, Twitch is a popular solution, controlling all data transmitted through the platform. This cannot be implemented in any of the previously discussed frameworks, as none of them natively support streaming of data directly. A partial expansion of this would be real time video communication. In this use case, the differentiation between producers and consumers of content is removed, as all active participants receive and send data to all others. This is a more complex case, as it not only requires the application to distribute each individual stream to all but one participants, it also needs to synchronize the streams to prevent common interferences in the flow of the conversation. Again, this is solvable in a client-server architecture, though it requires substantial resources to scale. However, it is not possible in the distributed frameworks, due to missing support for (synchronized) streams.

The third application is a file drop used for digital exam submissions in an educational context. A teacher distributes a digital exam sheet to students to fill out during a specified time frame. When they are finished, they send in their solutions, though they must not be able to overwrite their sheets once submitted. However, submitting multiple sheets per student is allowed. This acts as simple version control, giving the students more possibilities to work around possibly unstable internet connections and hardware and software failures on their end. Teachers may select the most recent usable file, while students have the option of requesting a grading of another file in case they suspect issues in the latest upload. A solution to this is possible in a client-server model, although reliable distribution of the sheets and authentication of the students could pose some challenges. The previously mentioned decentralized frameworks are insufficient for this use case, as none of them is able to forcibly distribute the sheets to the students in a reliable and accountable fashion, and they do not allow the teacher to configure a write-once location.

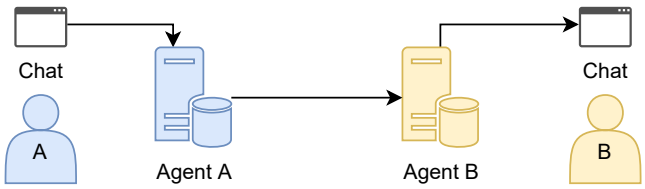

Figure 2: Sending a message in an application using persistent streams. Simplified workflow; coloring indicates ownership/control.

4 PERSISTENT STREAMS

To mitigate these issues we propose *persistent streams*, a communication protocol that augments the internet with ephemeral storage. It builds upon the idea of streams, – low-overhead data transmission between one source and one target – and removes the dependency on simultaneous availability of both parties. The protocol is aimed for use in a wide range of scenarios and applications, from resource-constrained devices in the Internet of Things to end-user applications that previously required a client-server based approach. Therefore, it must be implemented on top of – or integrating – a transport layer protocol such as QUIC streams [3]. This section describes the general idea and central concepts, as the development of the protocol is currently in progress and requires further expansion and refinement.

When applications use persistent streams, communication flows through *agents* controlled by their users, as depicted in Figure 2. In this example user A sends a message with a single recipient, user B, using a chat application. Similar to communication using E-Mail, agent A forwards the message to the agent of user B, where it resides until user B opens the chat application and retrieves the message. The agents may run on infrastructure the users control or as a service in a cloud, depending on each user's choice.

By using these agents as proxies, the protocol enables completely decoupling the distribution of the application from user communication and data storage. The provider of the application only handles distributing the application itself, without needing to provide additional resources to handle communication. Since the agents are responsible for data distribution, the protocol supports first-class group communication and allows scalability by design. This way, the single point of failure for the whole network, which is present in client-server models, is removed and replaced by a distributed structure where a node failure only impacts a single user's communication, as long agents are independently hosted.

However, in contrast to E-Mail, persistent streams also support streaming data, enabling their use in a multitude of

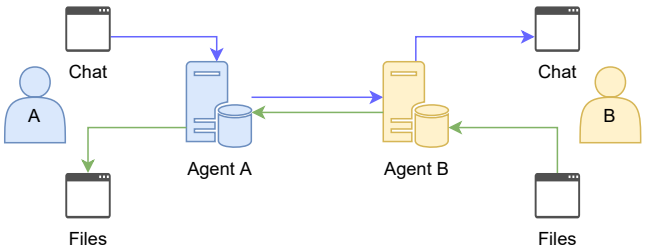

Figure 3: Abstract depiction of communication in two applications using persistent streams. Coloring indicates ownership/control.

additional scenarios. Agents are also agnostic to both the applications a user utilizes to access them as well as the kind of data that is transmitted through them. While in the example above, user B may have obtained the application from the same provider as user A, they could also use another (e.g. web) application using the same data structure. As Figure 3 shows, an agent is also not bound to a single application. In this example, while user A sends a message to user B, user B shares a file with user A. The agents handle all transmission and storage of data.

The core idea of the protocol is to enable asynchronous communication on an internet level, without specific applications needing to solve this issue independently. To do this, data may be *persistent* or *ephemeral* on either agent. Persistent data is accessible as long as it is present, while ephemeral data is only accessible once per user. The source agent is additionally able to distinguish which data is kept private and which is served to recipients.

In the examples above, the applications are *clients*. While they are actively operated by the user, this is not a requirement. A client may also be an online service that acts autonomously on behalf of the user. Either way, each client must be actively authorized by the user to access certain parts of their data. Clients inform an agent of their identity, which could be used to indicate a specific application or a certain (social) network. This way, clients may also be addressed directly, indicating to the agent which application a stream should be pushed to.

Streams are cohesive segments of binary data with minimal metadata and constitute the basic data unit of the protocol. A stream can be either discrete (enclosed data) or continuous, and only be modified by the user it was created by. Agents only keep minimal metadata for each stream, and streams cannot be stored or addressed on their own. They reside in *bundles*, which are sharable groups of streams enriched with extended metadata. Each bundle has an identifier that is unique for the user owning it, and recipients of the bundle can be defined explicitly. Once the recipients are defined or modified, the agent informs each of them (once), so they can accept or reject the bundle. All recipients that have accepted the bundle will receive updates to any stream inside that bundle pushed to their agent automatically. This scheme reduces communication overhead by *accepting* only once per bundle per recipient instead of each stream independently. Since all data inside streams is handled as binary and the user can decide which bundles a client may access, the protocol inherently allows for arbitrary interoperability between services. They only need to be able to parse and interpret the data received from the agent and be authorized as a client by the user.

Addresses in persistent streams are based on the scheme used in E-Mail. This should be both intuitive for most users, as well as practical, since it allows for utilization of the existing DNS to locate agents. However, persistent streams requires another optional field to specify a client application so it can be addressed directly. While further research is required to determine the final scheme, the basic structure is:
`[<client>:]<user>@<agent>`
Since this addressing scheme should feel familiar to most users, they should be able to exchange addresses in the same ways used currently. Additionally, since persistent streams is a data structure agnostic protocol, it can also be leveraged to host user directories. Hosting a client at a well-known location, where users can opt in to be listed in the directory, allows other users to discover them. This seems most practical on an application- or community-level, where the address of the directory would be included in the application or distributed publicly.

4.1 Application

While the protocol is still in early design stages, we can show some drafts for solutions to the use cases in Section 3. For example, these basic functionalities already support a simple microblogging application. The application creates a bundle per group of recipients and pushes messages as individual streams to distribute them among followers. In Figure 4 we call this application *decentralized social network* (DeSN). User A creates a post addressed specifically to DeSN clients of users B and C and pushes it to their agent in a stream. The agent distributes the post to the recipients' agents, who in turn forward the data to the target applications. While user C simultaneously uses another unrelated application utilizing persistent streams, that application is unaware of this post, as the agent only forwards the stream to DeSN. It is also possible to provide a timeline of public posts in the same way user directories can be implemented: A user sends an opt-in to a client at a well-known address. This client then in turn collects and lists public posts of this user for others to

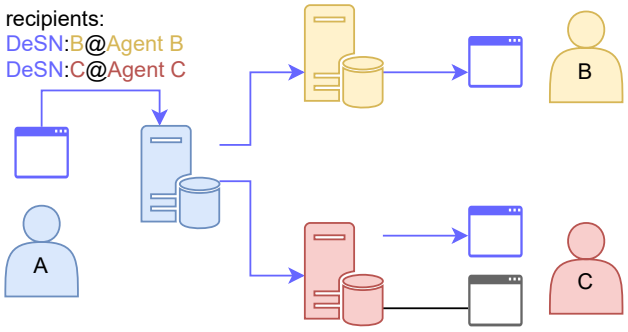

Figure 4: User A creates a post in a microblogging application (DeSN). Users B and C receive the post. Only the relevant application of user C receives the data.

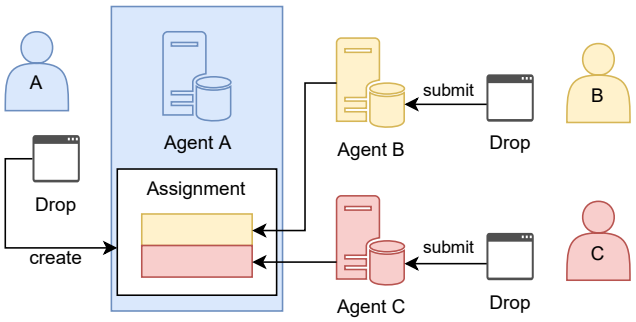

Figure 5: Teacher A creates a bundle and assigns write permissions for one new stream to each of the students (B and C). After B and C submit their solutions they are no longer able to edit them.

find. Other users can query this client for posts and receive the related addresses and bundle IDs.

The public streaming use case could be implemented similarly to this. However, the group video chat application additionally requires strict synchronization of streams using timestamps, where one participant's agent could be designated as authoritative, determining the order of stream frames. The end user applications could then synchronize their state to this using local lag as described in [6]. Provided that agents are hosted on reliable servers, re-synchronization should not be required often. To support the file drop application, the creator of a bundle needs to be able to allow students to create streams in their bundle. Figure 5 shows how a teacher (A) could allow students (B and C) to submit their solutions into a bundle that is under the teacher's control. To prevent editing uploads, the teacher must be able to define that newly created streams are internally persistent and externally ephemeral. This way students would only be able to upload solutions once, without being able to change their submissions afterward. A solution to achieve this in persistent streams are owner-definable default permissions for newly created streams in a bundle. When the students upload their solutions, the permissions are automatically applied so they are no longer accessible to anyone but the teacher.

In all of these scenarios, the application providers need to contribute distinctly fewer hosting resources in order to maintain the networks between users of their services than would be required in fully client-server based solutions. Since the data is under the users' control, they may also choose whichever user interface they like, as long as it supports the data format.

5 EFFICIENTLY HANDLING DATA STORAGE

Funneling all communication and user data through the agents controlled or owned by their users helps increasing privacy and data sovereignty of the users. However, that means agents can be bottlenecks for all communication of a user.

In contrast to current network infrastructure, which generally only forwards or drops traffic, agents must be able to (temporarily) store data so it can be distributed at a later time. Data that was received and acknowledged by an agent must not be lost unless the user instructs it to. To achieve this, agents require non-volatile storage to prevent data loss in failure scenarios. However, non-volatile storage has traditionally been too slow to handle the large amounts of incoming and outgoing network traffic present in busy infrastructure. This required advanced algorithms and complex architectures to balance both speed and reliability.

Non-volatile main memory (NVRAM) changes this, as it provides response times and throughput closer to DRAM than to solid state storage, while being resistant to power loss. [5] However, machines hosting busy agents might still require storage capacities in excess of what is practically available to them. We expect a few factors to influence this issue in the agent-specific use case positively:

(1) Firstly, real-time communication can be expected to require no longer caching than needed to receive and forward the data to their destination. This allows agents to discard this data quickly.

(2) Furthermore, data that is cached for long periods of time can be assumed to not be time-critical. As such, pushing it down in the storage hierarchy towards slower but larger storage tiers is a feasible solution if it has not been accessed for some time.

Adding NVRAM lowers the requirements on algorithmic solutions to fault tolerance significantly in this use case, as data that is not directly forwarded can be moved into non-volatile regions quickly. Given the first assumption above, it seems reasonable to expect systems to be able to hold enough data in NVRAM to cache communication for at least a few minutes. Even if all data would have to be cached in NVRAM, we could start moving the oldest blocks down another tier in the hierarchy. Following the second assumption, data that was not retrieved in the first minutes can be pulled from slower storage later without large impact on the requesting application.

Even though these are just general assumptions, we believe they provide an indication of feasibility to accelerate the storage system in a way that reduces the negative impact of the system on speed in regards to user experience. While this is not the only part showing possible optimizations, it is a central element with large consequences, especially if fault tolerance is required.

6 CONCLUSION AND OUTLOOK

In this work we explained how current approaches to asynchronous communication based on client-server models place large burdens on providers, how existing decentralized approaches fall short of general applicability, and how a novel approach – *persistent streams* – could provide a path to solve these issues in an efficient, resilient and privacy-preserving manner.

However, further research is required to ensure that persistent streams are generally usable as a protocol for most common use cases in distributed communication. For example, to ensure that interactive applications work without degradation through higher latency, a proof of concept needs to be built in order to evaluate real-world impacts of the changed routing. In this context we are currently investigating ways to minimize delays in processing on the agent. It should also be investigated whether it is possible to reduce overhead in storage by allowing a single stream to be part of multiple bundles. Furthermore, investigation is needed to determine if recipients should be able to select which streams in a bundle they are interested in, or whether proper utilization of bundles to separate streams is sufficient.

Overall we believe that the general structure of persistent streams is better fit for the task of solving decentralization and decoupling of user data from applications than existing approaches portrayed in Sections 2 and 3.

REFERENCES

[1] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR* (2014). http://arxiv.org/abs/1407.3561

[2] Long Cheng, Fang Liu, and Danfeng Yao. 2017. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 5 (2017), e1211.

[3] Jana Iyengar and Martin Thomson. 2021. *QUIC: A UDP-Based Multiplexed and Secure Transport.* Technical Report 9000. https://doi.org/10.17487/RFC9000

[4] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. A Demonstration of the Solid Platform for Social Web Applications *(WWW '16 Companion)*. International World Wide Web Conferences Steering Committee, 223–226. https://doi.org/10.1145/2872518.2890529

[5] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 288ž2013303. https://doi.org/10.1145/3357526.3357541

[6] Sebastian Schuster and Torben Weis. 2011. Enforcing Game Rules in Untrusted P2P-Based MMVEs. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 288–295. https://doi.org/10.4108/icst.simutools.2011.245550

[7] James M Snell and Evan Prodromou. 2017. *Activity Streams 2.0.* Technical Report. W3C. https://www.w3.org/TR/2017/REC-activitystreams-core-20170523/

[8] Maximilian Uphoff, Matthäus Wander, Torben Weis, and Marian Waltereit. 2018. SecureCloud: An Encrypted, Scalable Storage for Cloud Forensics. IEEE, 1934–1941. https://doi.org/10.1109/TrustCom/BigDataSE.2018.00294

[9] Christopher Lemmer Webber, Jessica Tallon, Erin Shepherd, Amy Guy, and Evan Prodromou. 2018. *ActivityPub.* Technical Report. W3C. https://www.w3.org/TR/2018/REC-activitypub-20180123/

[10] Torben Weis and Arno Wacker. 2011. Federating Websites with the Google Wave Protocol. *IEEE Internet Computing* 15 (2011), 51–58. https://doi.org/10.1109/MIC.2011.28