SocioPath: Protecting privacy by self-sufficient data distribution in user-centric networks

Fabian Hartmann and Ingmar Baumgart
Institute of Telematics
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

***Abstract*—Communication between users poses a privacy problem as soon as it relies on a third party. The problem ranges from personalized advertising to mass surveillance and applies primarily to centralized application service providers. However, also nodes in decentralized approaches are often under control by a third party. We argue that application data and metadata between a set of users can stay private only by self-sufficiency, i.e. exclusive end-to-end communication between devices under these users' control. State synchronization is a main challenge here, since especially mobile devices are prone to churn and varying connectivity. We present SocioPath, a decentralized protocol for self-sufficient user-to-user communication. It handles device heterogeneity by decoupling data objects from notifications and keeping recurrent state exchanges small. Additionally, it offers a user-centric application interface which abstracts from devices towards the user. Evaluation results show that even under heavy churn, it is possible to achieve a high delivery ratio and large scalability.**

I. Introduction

Application service providers like Facebook or Google are popular instruments for communication between users. A user does not perceive such a provider as a participant in the communication. Instead, the provider is merely a means to an end by providing large data storages with high availability. Hence, transferring data from the users' personal devices to the provider is accepted as a necessary evil here. Full access to all customers' data enables a third party (such as the provider or an intruder) to perform large-scale analysis on the customer data and jeopardize the customers' privacy. Possible analysis goals are detailed user profiling, industrial espionage or mass surveillance by authorities.

One straightforward solution for the customer is end-to-end encryption. This prevents a third party to read the data two users exchange. Also, connection metadata such as IP addresses can be obfuscated by using proxies or an onion routing service (e.g. [1]). However, the users still have to give away their *application metadata* to the provider. This type of metadata includes two types of information: Firstly, *explicit metadata* is required for the service to address its users and to function properly. Examples are email addresses, online social network profiles or entries in an address book. Secondly, *implicit metadata* accumulates at the provider's end when users access the service. Examples are the date and time of a sent email or the frequency of visiting a social network profile. Over time, a third party can at least create profiles about inter-contact times and contact durations between specific users. With further research via side channels (such as performing a web search on an email address), it might infer even more information about a user. The impact of implicit metadata on privacy is subject to current research [2].[1]

This problem also applies to decentralized approaches for user-to-user-communication. Such approaches make use of federated servers (e.g. [3]) or structured P2P networks (e.g. [4],[5]). Unless each user hosts his data on a separate dedicated server, a federated server holds multiple user data sets, similar to a centralized approach. In a structured P2P network, all data gets distributed across the participating devices according to a protocol-specific metric. A single user cannot influence if the device which is responsible for his data belongs to himself, a friend or an unknown person.

We argue that data and application metadata between a set of users can stay private only if exclusively devices under these users' control communicate end-to-end. The devices controlled by a single user range from personal devices like smartphones and PCs to high-bandwidth servers, as long as they are administrated by the user. The amount and types of such devices differ from user to user. This heterogeneity has a strong impact on data availability, yet only these devices should realize all data storage and distribution. This *self-sufficiency* is a novel approach for decentralized user-to-user-communication.

Self-sufficiency needs to be defined per communication process: If Alice, Bob and Charlie share pictures among each other in a group, all their devices can be involved in the distribution of the pictures. If Bob wants to send an instant message to Charlie only, Alice's devices must not get involved in the transmission or the storage of the message, since that would pose an unwanted leakage of data and metadata.

These examples go intuitively together with a *user-centric* approach: Both application developers and end users should only have to care about the addressing of human recipients, not their specific devices.

We present *SocioPath*, a decentralized protocol with the following main features:

- self-sufficient communication between two or more users in a closed group
- user-centric abstraction from single devices
- an exchangable and adaptable distribution strategy
- handling of temporary device unavailability (churn)

[1] http://metaphone.me
https://immersion.media.mit.edu

SocioPath is based on a user-centric publish/subscribe paradigm, where each topic has a well-defined owner who has full control over the access rights of other users. Section II of this paper describes the basic concept and Section III elaborates on the protocol details. The protocol relies on some kind of distribution strategy, which distributes published data objects between the involved devices. An example distribution strategy is described in Section IV. Evaluation results in Section V show that even under heavy churn, it is possible to achieve a high delivery ratio for each published object. These primary results are followed by some application examples (Section VI). Section VII shows an overview about related work. The paper closes with a summary and considerations towards future improvements (Section VIII).

II. Basic Concept

SocioPath offers a publish/subscribe (pub/sub) service for the distribution of a data *object* from one user to a closed group of *recipients*. Each object is tied to a specific *topic*, whereas each topic is tied to a specific user, the *topic owner* (TO). Every user has a unique user ID (*userId*) and every topic has the following structure:
$Topic :=< Title@userId_{TO} >$. *Title* is an arbitrary string and identifies an application or service that relies on SocioPath for data distribution.

Let the set of all users in a SocioPath network be $\mathbb{U}$. The set of all topics in the system amounts to $\mathbb{T} := \bigcup_{\forall X \in \mathbb{U}} \mathbb{T}_X$ with $\mathbb{T}_X$ being the set of topics where user X is TO. $\forall X, Y \in \mathbb{U}$ with $X \neq Y$ it is $\mathbb{T}_X \cap \mathbb{T}_Y = \emptyset$.

TO X of $t \in \mathbb{T}_X$ defines the *allowed subscriber* set $\mathbb{A}_t$ from his contacts $\mathbb{C}_X$. Only users in $\mathbb{A}_t$ may learn about and retrieve an object α_t tied to t. $\mathbb{S}_t$ is the set of *subscribers*, i.e. users that make use of this permission. They get notified about a new or modified α_t. Hence, it is $\mathbb{S}_t \subseteq \mathbb{A}_t \subseteq \mathbb{C}_X \subseteq \mathbb{U}$. $\mathbb{S}_t$ includes at least the TO himself. Let $\mathbb{D}_X$ be the devices that user X possesses. α_t is to be distributed to the devices $\mathbb{D}_t \subseteq \bigcup_{\forall X \in \mathbb{S}_t} \mathbb{D}_X$.

TO X of $t \in \mathbb{T}_X$ also defines the *allowed publisher* set $\mathbb{P}_t \subseteq \mathbb{C}_X$ of users allowed to publish an object α_t. $\mathbb{P}_t$ includes at least the TO himself. If user Y publishes α_t, X accepts and distributes α_t to $\mathbb{S}_t$ only if $Y \in \mathbb{P}_t$.

This concept of data distribution involves only *target devices*, i.e. devices that are under control by the users in $\mathbb{P}_t \cup \mathbb{S}_t$. The TO devices coordinate the data distribution from $\mathbb{P}_t$ to $\mathbb{S}_t$ devices. This results in decentralized and self-sufficient data distribution which protects both data and metadata privacy of the involved users: If α_t is distributed between target devices only, only the device owners gain knowledge that α_t exists. We call this feature *closed data invariant*. Figure 1 depicts an example scenario.

For maximum privacy, only the TO of a topic t knows about $\mathbb{S}_t$, $\mathbb{P}_t$ and $\mathbb{A}_t$. Another possibility is to distribute this knowledge to target devices of other users, which poses a trade-off between availability and privacy where e.g. subscribers can learn about other subscribers. However, the closed data variant still holds.

Enforcing the closed data variant is more restrictive than in other pub/sub systems, where a third party broker maintains

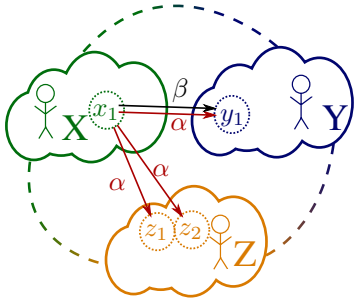


Fig. 1: Closed data invariant: The recipients for object α are both Y and Z, while object β is for Y's eyes only. If each object is transferred between devices of the respective target device group only, Z never learns that β exists.

all subscribers and published messages. In a structured P2P network, a node's responsibility for an object is defined by the closeness between its nodeId and the object according to a protocol-specific metric. If a broker node is defined by its nodeId only, this node might not be a target device. This violates the closed data invariant.

User-controlled devices differ strongly in terms of availability and connectivity: A smartphone might be switched off at night and have slow or expensive connectivity during most of the day. A personal server might have 99,9% availability and neither traffic nor bandwidth are a big issue. To deal with heterogeneity, we separate the distribution of an object into two steps:

- First, all target devices receive a *notification*. The notification contains information like the object's bytesize and its source devices. We assume that notifications are small enough that even devices with slow or expensive connectivity can cope with them.
- Second, each target device decides if and when it wants to *retrieve* the object it was notified about. Since a data object can be very large, a mobile device with slow connectivity can delay the retrieval until it has WiFi access. If no suitable application is running on a target device, it can even ignore the notification.

Additional notes

All objects are persistent until an individual time-to-live (TTL). If a new user subscribes to a topic, he still has access to older objects that do not have exceeded their TTL yet.

The distribution of a notification to all target devices is the task of a replaceable *distribution strategy* that can be freely defined and improved in terms of efficiency and resource-awareness [6], as long as it holds the closed data invariant. For example, notifications can be delegated to a stronger device among the target devices. Furthermore, the strategy needs to cope with offline devices. Missed notifications have to be resent to devices as soon as they are available again. We present an example distribution strategy in Section IV.

The grounds for a target device's decision if and when to retrieve the object is to be defined as part of the distribution strategy as well.

Devices that have successfully retrieved an object can provide it themselves to other target devices. This requires in turn keeping a list of source devices up-to-date and making it available for other target devices. It is possible to reuse SocioPath's pub/sub service here, as we do for maintenance information (see Section III-D). However, the details for multi-source retrieval are out of scope of this paper. For the remainder, we assume that objects are retrieved from their origin or a TO's device respectively.

III. Protocol Details

In this section we discuss SocioPath's protocol details that are required to establish the basic concept we described. The presented design features the overall architecture, consisting of the application interface, RPC message exchange and the required data structures. In addition, we describe required workflows such as adding a new device or a new contact. This design is valid for all distribution strategies, for which we present a basic example in Section IV.

A. Application interface

SocioPath is designed to be a protocol that supports multiple applications per device at the same time. To achieve this, each application assigns its objects to a different topic. Each object is uniquely identified by the combination of topic, a unique object ID (*objId*) and creation timestamp. It gets passed to SocioPath, which in turn is responsible to notify target devices and establish retrievals on request. The pub/sub interface is shown in Figure 2 and provides methods as follows. Note that methods from the application to SocioPath are symbolized by → and methods from SocioPath to the application are symbolized by ←.

- → **publish**(*topic, objId, timestamp, object, ttl*)
  This method passes an object to SocioPath. From the application point of view, SocioPath offers a best-effort service to notify all target devices about the new object and deliver it if demanded. The object gets stored in SocioPath's data storage until *ttl* exceeds.
- → **subscribe**(*topic*)
  This method subscribes the owner of this device to *topic*.
- → **unsubscribe**(*topic*)
  This method unsubscribes the owner of this device from *topic*.
- ← **notify**(*topic, objId, timestamp*)
  After a target device was notified about a new object, this method notifies all applications bound to *topic* about that object.
- → **retrieve**(*topic, objId, timestamp*)
  This method lets an application retrieve an object it was notified about via **notify()**. It is transparent to the application if the object is already available on the same device or if it has to be requested by SocioPath from another device first.

B. Signaling between two devices

All signaling communication in SocioPath is RPC based. We assume for the remainder of this paper that all communication is encrypted and authenticated using public key cryptography. The RPC interface offers services as follows. Note that requests are symbolized by → and responses are symbolized by ←.

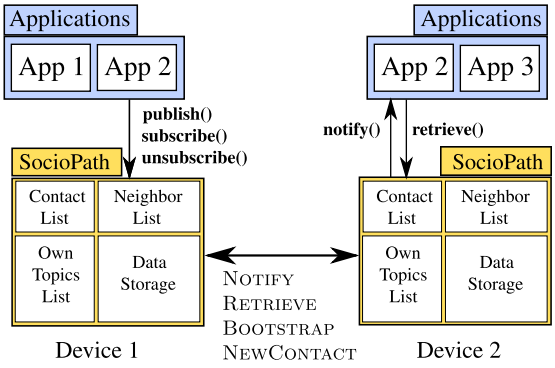


Fig. 2: Architecture overview

- → NOTIFY(*topic, objId, timestamp, publisher, objSize, obj*)
  This request notifies a receiving device about a new or modified object. The publish timestamp lets the receiver detect if an existing object has been modified. Additionally, the request holds the original publisher, since he might differ from the owner. It also holds the object's size, since this is an important factor for the RETRIEVE decision. Furthermore, a small object can be piggybacked as *obj* if its size does not justify the overhead of a decoupled retrieval.
- ← NOTIFY()
  This response is an acknowledgment without further information.
- → RETRIEVE(*topic, objId*)
  This request lets a device retrieve an object it was notified about. A retrieval is not necessary if the object was piggybacked in the *obj* field of a NOTIFY request.
- ← RETRIEVE(*object*)
  This response holds the object that was requested for retrieval. As soon as it has arrived, an application can **retrieve()** it from the local device.
- → BOOTSTRAP()
  This request is triggered by a new device to be added into a user's device pool (see Section III-E).
- ← BOOTSTRAP(*userId, publicKey, encPrivateKey, contactList, neighborList, ownTopicList*)
  This response holds all information to bootstrap the new device (see Section III-E).
- → NEWCONTACT(*userId, publicKey, userDevices*)
  This request holds the information that is sent to a new contact during the pairing process (see Section III-F).
- ← NEWCONTACT(*userId, publicKey, userDevices*)
  This response holds the information that is back sent from a new contact during the pairing process (see Section III-F).

C. Data Structures

Each device needs to hold multiple data structures for maintenance. The contents of the data structures are different for

userId	User Alias	Public Key	Private Key
0xAAAA	Alice Alison	dX38UjmI...	KmL71BgN...
0xBBBB	Bob Bobsen	lsuusFnc...	
0xCCCC	Charlie Charleston	BQ7RKNUSe...	

TABLE I: Example Contact List for user Alice

userId	deviceId	Device Locator	Comment
0xAAAA	0x1111	123.45.67.89	Alice's device
0xBBBB	0x2222	234.56.78.90	Bob's device
0xCCCC	0x3333	34.56.78.123	Charlie's 1st device
0xCCCC	0x4444	45.67.89.234	Charlie's 2nd device

TABLE II: Example Neighbor List for user Alice

Topic	objId	Timestamp	TTL	Object
Profile@AAAA	23	1234	-1	(bin)FF3C...
Profile@AAAA	42	1234	-1	July 1st...
InstMsg@AAAA	1208234	1000	5000	Hi, how...
Files@BBBB	1298912	1800	-1	*(not retrieved)*

TABLE III: Data Storage on a device of user Alice

Topic Title	Allowed Publishers	Subscribers
OwnContacts	Topic Owner	Topic Owner
OwnTopics	Topic Owner	Topic Owner
Devices	Topic Owner	Topic Owner + Contacts
SubscribeMe	Topic Owner + Contacts	Topic Owner

TABLE IV: Maintenance Topics

each user but synchronized across all devices that belong to a single user. Synchronization is realized via notifications about maintenance topics (see Section III-D) and hence the task of the distribution strategy. We aim for *eventual consistency* [7], i.e. each device makes its decisions on its current state and updates are assumed to propagate eventually. The data structures for a specific user are:

1) Contact List: The Contact List contains the user X who controls the device and $\mathbb{C}_X$. Each entry is uniquely identified by a 160 bit userId, which is the only user information required by the protocol to function properly. Additional fields are a human readable alias and a public key for encryption and authentication. The process of adding a new contact is described in Section III-F. An example Contact List is displayed in Table I.

2) Neighbor List: The Neighbor List holds all required information about the X's and $\mathbb{C}_X$'s devices. This includes a mapping between userId and a device ID (*deviceId*) plus a device locator (e.g. IP address). The deviceId gets randomly generated once the device joins for the first time. An example Contact List is displayed in Table II. For the remainder of this paper, we assume that the problem of locating a device is solved as long as they are online.

3) Own Topics List: The Own Topics List on a device of user X holds information about all topics $\mathbb{T}_X$. Each topic is identified by its title. The access rights per topic t are defined by three lists which hold the userIds in $\mathbb{P}_t$, $\mathbb{A}_t$ and $\mathbb{S}_t$ respectively.

4) Data Storage: The Data Storage holds the notifications for topics the user is subscribed to as well as corresponding objects if available. The value of objId is either chosen randomly or set to a fix value by the application during **publish**. An application can overwrite an existing object by publishing a second time using the same objId.

An example Data Storage is displayed in Table III: Here, Alice's profile page in a social network application has the topic *Profile@AAAA* and features different objects, such as a profile picture (objId 23) and a birth date (objId 42). If Alice updates her picture, this triggers a publish with topic *Profile@AAAA*, objId 23 and a new timestamp. Each device that is notified about this new publish, compares the new timestamp with the one in the Data Storage and sees the object has been updated and can be retrieved.

D. Maintenance Topics

Pub/sub is not only useful for applications, but can also be used to keep the data structures from Section III-C up-to-date. For example, if a user wants to subscribe to a specific topic, all devices of the TO must be informed. We can achieve this by defining maintenance topics that use the same pub/sub service that is offered to applications. The maintenance topics are defined per user, hence every user is the TO of his own maintenance topics. The maintenance topics for user X are:

- *OwnContacts@X*: For updates in X's Contact List. Examples: New contact added, existing contact is removed. See Section III-F about adding a new contact.
- *OwnTopics@X*: For updates in X's Own Topics List. Examples: New subscriber, new topic, revoking a contact's publishing rights from an existing topic.
- *Devices@X*: For updates in devices X owns. The subscribers update their Neighbor Lists accordingly. Examples: New device added, IP address change.
- *SubscribeMe@X*: For subscription requests by other contacts. The topic to be subscribed is piggybacked in the notification. See Section VI for an example use case.

Depending on the maintenance topic, updates are only for the TO himself or for the TO and his contacts. The Own Topics List gets filled accordingly. For example, if the TO gets a new device, his contacts must be informed about this, so notifications about new objects reach the new device as well. However, if the TO revokes a contact's publishing rights for a topic, this should be only propagated to his own devices. Details are displayed in Table IV.

E. Adding new devices

In this section we discuss how multiple devices get assigned to a single user X. The very first device in a device pool sets up the data structures with initial values. X's contact information as described in Section III-C1 gets created and is the only entry in the Contact List. Similarly, the device x_1's information as described in Section III-C2 is the only entry in the Neighbor List. The Own Topics List contains only the maintenance topics with full access rights given to X as the TO. The Data Storage is empty.

Additional devices are added incrementally into the user's device pool. Each new device x_{N+1} has to be paired with an existing device $x_i \in \mathbb{D}_X =: \{x_1, \ldots, x_N\}$ from X's device pool. x_{N+1} triggers the pairing by sending a BOOTSTRAP request to x_i. In response, x_i fully transfers its Contact List, Neighbor List and Own Topics List to x_{N+1}. The Contact List also contains the private key of X which is symmetrically encrypted with a user-provided passphrase. The Data Storage is not transferred: It could be very large in size and not all objects are relevant for x_{N+1} if it runs other applications. Instead, we rely on the distribution strategy, its churn handling (see Section IV-B) and x_{N+1}'s retrieval choices during the regular protocol cycle. This will take more time to transfer all relevant objects to x_{N+1}, but it is more efficient in terms of traffic.

Finally, x_{N+1} publishes its own device information to topic *Devices@X*, which informs $\{x_1, \ldots, x_N\}$ about the new device. Since $\mathbb{C}_X = \mathbb{S}_{Devices@X}$, all contacts' devices get informed as well.

F. Adding contacts

If two users X and Y want to add each other mutually to their Contact List, one device x_i of user X has to pair with a device y_j of user Y. Pairing two devices by different users requires the same mechanism as with two devices for the same user, but less information is exchanged: Here, only information about the user himself (such as the userId and his public key) and the information about the own devices get exchanged. The latter is a subset of the Neighbor List, which only contains the entries that are mapped to the own userId.

x_i enters Y into its Contact List and the devices $\{y_1, \ldots, y_M\}$ into its Neighbor List. It publishes this information to *OwnContacts@X*, which notifies all of X's other devices $\{x_1, \ldots, x_N\}$. These also update their Contact and Neighbor Lists, hence every device of X knows about the new contact Y and his devices. Device y_j proceeds in an analog way.

IV. DISTRIBUTION STRATEGY

In this section we describe an approach for distributing NOTIFY messages to target devices. It handles temporary unavailability of devices via an additional message type named STATE. This message type enables devices to detect missing notifications and trigger retransmissions. We evaluate an implementation of this strategy in Section V.

A. Procedure

0) When user X at device x_1 publishes a new object from an application to SocioPath, the procedure checks the object's topic t first. Let the TO be user O. If $X \neq O$, the procedure continue with step 1. If $X = O$, it continues with step 4.
1) Since X is not TO, the TO's devices $\mathbb{D}_O =: \{o_1 \ldots o_N\}$ have to be notified first. x_1 looks up all known TO's devices in the Neighbor List and x_1 sends a NOTIFY to each of them.
2) Each notified device $o_i, 1 \leq i \leq |\mathbb{D}_O|$ checks in its Own Topics List if $X \in \mathbb{P}_t$. If true, o_i has to retrieve the object first. o_i instantly sends a RETRIEVE request to x_1. If $X \notin \mathbb{P}_t$, o_i discards the notification and the procedure is aborted. Note that two devices o_i and o_j might decide differently due to inconsistent $\mathbb{P}_t$ states.
3) The object is sent from x_1 to the retrieving devices in $\mathbb{D}_O$. The procedure is stalled on the retrieving devices until the object was successfully transmitted.
4) The object is stored on a device of O, so it notifies all target devices. It looks up $\mathbb{S}_t$ in the Own Topics List. For each subscribed user S, it looks up all devices $\mathbb{D}_S$ from the Neighbor List and sends a notification to all of these devices.
5) Each notified target device can now retrieve the object from the TO's device that sent the NOTIFY.

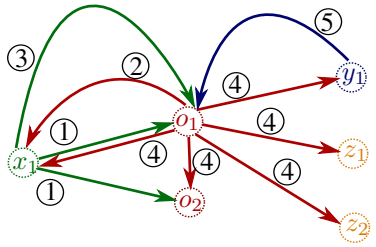


Fig. 3: Distribution strategy procedure

The procedure steps is depicted in Figure 3. Here, o_1 decides to retrieve the data object from a_1, while the procedure is aborted on o_2 due to a different $\mathbb{P}_t$ state.

Note that target devices may receive multiple notifications per published object. Such duplicates with same topic, objId and timestamp are ignored.

B. Handling Churn

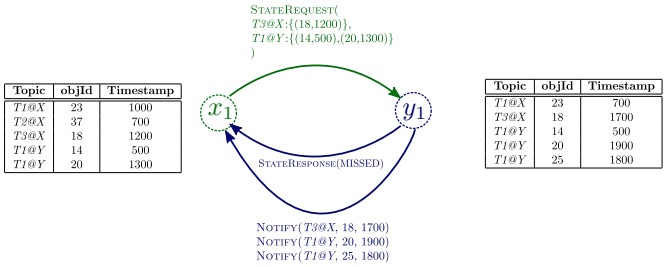
The procedure as described above assumes that all involved devices are permanently available. To handle intermittent unavailability (*churn*), each device can send a STATE RPC request to another device in its Neighbor List. By doing so, the requesting device asks the receiving device if it knows any notifications the requesting device has missed. STATE requests can be sent periodically or reactively, for the latter we assume that the device can detect whether it can reach other devices or is unavailable. Therefore it can detect when it returns from a period of unavailability and sending a STATE request is appropriate.

The content for each STATE request is defined by the target device's owner. Hence, one device sends a copy of the same STATE request to all devices of the same user.

The device x_1 of user X creates a STATE request for the devices of user Y as follows: It iterates over its Data Storage and collect all publishes with *relevant topics*, i.e. the topics where

- X is subscribed to and Y is TO
- If $X \neq Y$: X is TO and $Y \in \mathbb{P}_t$ according to Own Topics List on x_1

For each of these publishes, a tuple <*topic*, *objId*, *timestamp*> is hashed into a bloom filter [8]. The resulting payload of the



Topic	objId	Timestamp
T1@X	23	1000
T2@X	37	700
T3@X	18	1200
T1@Y	14	500
T1@Y	20	1300

Topic	objId	Timestamp
T1@X	23	700
T3@X	18	1700
T1@Y	14	500
T1@Y	20	1900
T1@Y	25	1800

Fig. 4: Example STATE request/response process: Device x_1 asks y_1 if there are new or updated notifications and y_1 responds accordingly. The only relevant topics here are *T3@X* where user Y is allowed to publish and *T1@Y* where user X is subscriber.

STATE request is the bloom filter plus a list of the relevant topics. x_1 sends this to all known devices of Y.

Each target device of Y, here y_1, iterates over the topic list from the request. For each topic, the tuples of all known publishes from the Data Storage are generated as above. y_1 checks if the received bloom filter contains each tuple. If a check fails, y_1 has detected a publish which x_1 is unaware of. In this case, y_1 sends a NOTIFY about the corresponding data item call to x_1. x_1 updates its Data Storage accordingly. Figure 4 shows an example message exchange.

Note that this whole process is unidirectional, due to the nature of bloom filters. y_1 cannot detect if x_1 has inserted publishes into the bloom filter that y_1 is unaware of. This has to be detected by a STATE request from y_1 for X.

V. EVALUATION

In this section, we evaluate an implementation of the presented SocioPath protocol and distribution strategy. In this evaluation, we focus on the distribution and retransmission of NOTIFY and STATE messages: Instead of using RETRIEVE requests, all objects are piggybacked in the notifications.

A. Test Setup

For the implementation, we use the overlay simulation framework OverSim [9]. For each simulation run, we generate a social graph based on the small-world graph generation algorithm by Watts and Strogatz [10]. A test application models typical instant messaging behaviour between users that are connected in this social network [11]. The underlying network is modeled by OverSim's SimpleUnderlay underlay abstraction, which is based on real Internet latency measurements.

The following general parameters are invariant for each simulation run: Each object is between 2 and 150 bytes size to reflect the character count of a typical instant message. On average, each device creates a message every 300s. All simulation runs last 10^5 seconds. The overall number of users is 200. Each user has on average 5 devices. An exponentially distributed churn model triggers a device's availability by switching it off (*deadtime*) and on (*lifetime*) successively. The average deadtime for each device is 1000 seconds.

The following parameters were evaluated. For each of the following sections in this paper, one parameter is chosen as a variable while the others keep their base values which are chosen as follows:

- STATE messages are sent reactively.
- The average lifetime for each device is 10^4 seconds.
- Each user is subscribed to 15 topics.

With the mentioned parameters being variable, we measured two performance indicators:

- *Notification delivery CDF.* It is difficult to define a success or failure for a notification transmission in our scenario: We accept and assume that devices can be unavailable for a long time. For this reason, we chose a empirical cumulative distribution function (CDF) that maps the delay from application publish until arrival at a target device on the x-axis. The CDF states that the probability for a notification to arrive at a target device in less than x seconds is $f(x)$. Dead devices are not simulated as switched off, instead they lack connectivity. This means that a device still can generate publishes during its deadtime, even if it cannot send them. This time is part of the delivery latency.
- *Bandwidth usage.* The second indicator is the overall sending bandwidth per device in bytes per second. This includes all outgoing UDP traffic.

B. STATE sending intervals

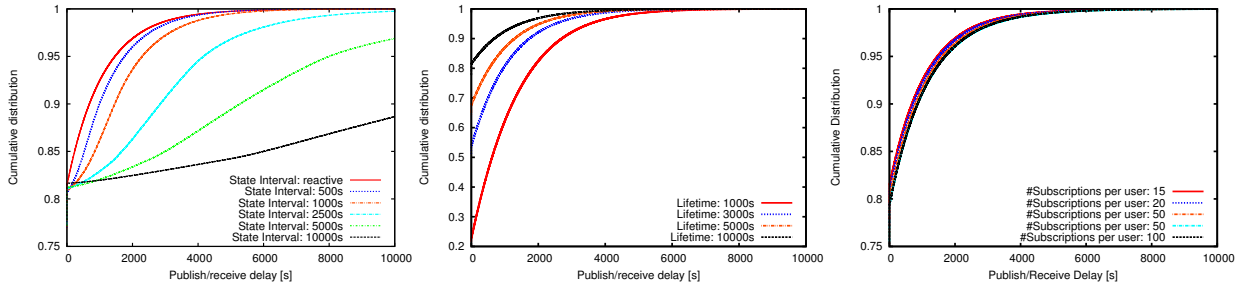
STATE messages are our approach for handling churn. We simulated both periodic and reactive behavior (see Section IV-B). For periodic behavior, we expected a strong impact of the average STATE sending interval (ssi) on both the notification delivery probability and the bandwidth usage. We expected reactive behavior to be the most effective and efficient, since a device uses knowledge about its actual availability here unlike in the periodic approach.

Figure 5a confirms these expectations: The reactive approach outperforms periodic STATE messages with $ssi = 500s$. For less frequent periodicities, the publish/receive delays are considerably longer, since it takes longer until STATE take care of missed notifications. For the reactive approach, about 90% of the notifications are delivered in less than 1000 seconds. An $ssi = 10000$ seconds still results in about 83% of the notifications being delivered in the same time, however the low gradient of the graph indicates that most of these were direct NOTIFY messages between online devices, with the STATE messages having only a small effect.

Figure 6a shows the bandwidth usage (y-axis) for periodic behavior i.r.t. different ssi values (x-axis). These are compared with costs for reactive behavior as a baseline, which only depends on the churn behavior of a device. Given the same churn behavior, the reactive behavior has much lower costs. Naturally, long intervals between STATE messages result in lower costs than the baseline, but results in intolerable publish/receive delays as shown in Figure 5a.

C. Impact of churn

Next, we set STATE sending to reactive behavior and varied the average lifetime of devices. Figure 5b shows the large

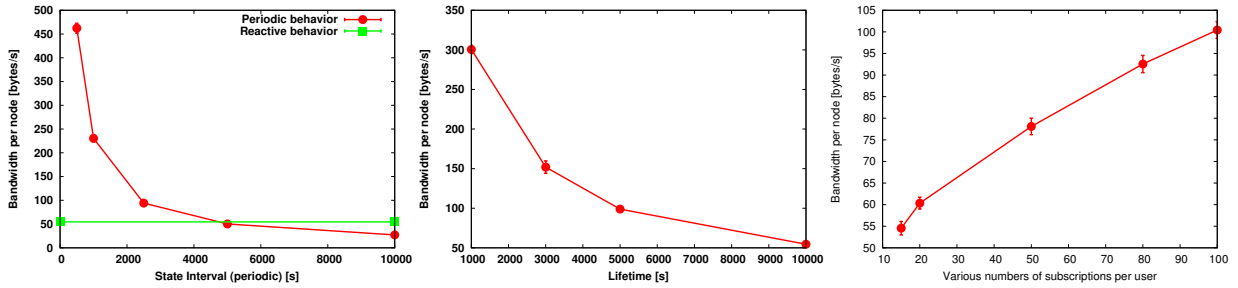


(a) Varying periodic STATE intervals vs. reactive behavior

(b) Varying device lifetimes

(c) Varying numbers of subscriptions per user

Fig. 5: Cumulative publish/receive delay distribution



(a) Varying periodic STATE intervals vs. reactive behavior

(b) Varying device lifetimes

(c) Varying numbers of subscriptions per user

Fig. 6: Bandwidth usage per node

impact of the lifetime: If a device's average lifetime equals the deadtime of 1000 seconds, it takes about 8000 seconds to deliver all messages. With an average lifetime of 10000 seconds, the same amount is delivered in less than 5000 seconds. With an higher average lifetime, most notifications are delivery instantly, indicated by the low gradient of the corresponding graph for larger times. However, even under heavy churn a delivery rate of 100% is reached due to the STATE messages.

Due to the reactive STATE behavior, the bandwidth usage drops with increasing lifetimes: Less STATE messages have to be sent and the overall higher availability of devices results in less resent NOTIFY messages.

D. Number of subscriptions

We varied the average number of subscriptions per user to increase the load for devices. Figure 5c shows that our protocol scales well i.r.t. publish/receive delays. More subscriptions only have a minimal impact on the delay since messages are delivered successively to more devices. Figure 6c shows that the bandwidth usage per device rises about proportionally with the average number of subscriptions. This is an intuitive result which one would expect with any other overlay architecture as well.

VI. APPLICATION USE CASES

In this section we describe three real-world application use cases and how they apply to the presented design. For clarity, we only describe the flow of notifications and assume that each device has retrieved the object in question before it can notify others.

A. File Sharing

Figure 7 depicts the following use case: Users X and Y play in a band and have recently welcomed Z as new member. X has created a topic for sharing song ideas via audio files and therefore is the TO for *SongIdeas@X*. Z uses one of his own devices, z_1, to subscribe to that topic (step 1). x_1 sees $Z \in \mathbb{A}_{SongIdeas@X}$ in the Own Topics List and enters Z into $\mathbb{S}_{SongIdeas@X}$. Earlier, X added Y to $\mathbb{P}_{SongIdeas@X}$. On Y's next publish of an audio file using y_1 (step 2), only x_1 gets notified, since Y has no information that Z is also subscribed. After retrieving the file (step 3), x_1 sees Z in $\mathbb{S}_{SongIdeas@X}$ and notifies z_1 and z_2 (step 4).

B. Instant Messaging

In the file sharing use case, we have one or a few publishers and possibly many subscribers. Instant Messaging works the other way around: The recipient of the message is the sole subscriber, but many contacts might be allowed to publish, i.e. message him. In order for X to send a message to Y, X has to publish to e.g. *InstMsg@Y*. Since only Y himself is subscriber for this topic, X's and Y's devices are the only target devices. If Z wants to send a message to Y, he also publishes to *InstMsg@Y* and again, Z's and Y' devices are the only target

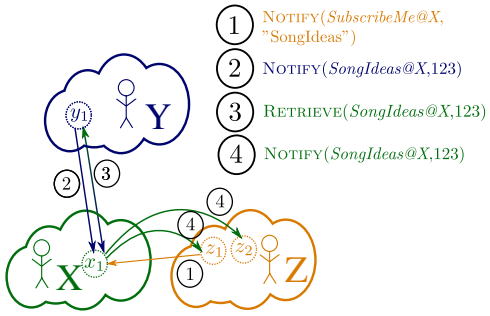


Fig. 7: File sharing use-case

devices and X is not involved in this communication process whatsoever. Since most instant messages are small in size, they might also be piggybacked in the *obj* field of the NOTIFY request, thus eliminating all retrieval overhead.

C. Smart Home

Smart home privacy is a recent research field [12], which is also applicable to our scenario. For example, a sensor which monitors X's fridge or mailbox could be another SocioPath-enabled user device that belongs to X. It publishes to a topic like *FridgeContent@X*. Here, X as the TO would be the only subscriber, optionally he could let other family members subscribe. However, he stays in full control about the access to the sensor and the data distribution stays self-sufficient.

VII. Related Work

Privacy in user-to-user-communication is a strong motivation for decentralized online social networks. In DHT-based approaches ([4], [5], [13]), all data is spread evenly across the participating nodes. If a node is responsible for maintaining a data object, it also has access to application metadata tied to the data object. However, a node's responsibility is not defined by its user ownership, but by a protocol-specific metric based on nodeIds. Depending on the network size, this gives away application metadata to a third party with a certain probability. Approaches that rely on federated servers ([3], [14]) are self-sufficient if and only if each user stores only his own data on his own server. In SocioPath, a server with high availability is helpful in a device pool but not required. In friend-to-friend networks and Darknets, each overlay hop is between devices that belong to users that trust each other. While this also applies to SocioPath, the application scenarios are different: In Turtle [15], search requests for e.g. a file are flooded across friend links, obfuscating source and destination of a request. In SocioPath, all objects are tied to a specific user who defines the recipient group and all recipients are aware of this owner.

VIII. Conclusion

We presented SocioPath, a privacy-preserving protocol for communication between multiple users in a closed group. Our self-sufficient concept enforces exclusive end-to-end communication between devices that belong to these users. This keeps all stored data and application metadata private from third parties. SocioPath handles device heterogeneity by decoupling data objects from notifications and keeping recurrent state exchanges small. Additionally, it offers a user-centric application interface which abstracts from devices towards the user.

However, device heterogeneity and varying connectivity are a major challenge in this scenario. We implemented and evaluated a basic distribution strategy which showed promising results, such as reliable delivery ratios and scalability with higher demands. For future strategies we aim at awareness and anticipation for device resources and user behaviour [6].

References

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th Conference on USENIX Security Symposium*, San Diego, CA, USA, Aug. 2004, pp. 303–320.

[2] B. Greschbach, G. Kreitz, and S. Buchegger, "The Devil is in the Metadata – New Privacy Challenges in Decentralised Online Social Networks," in *Proceedings of the 2012 International Workshop on Security and Social Networking (SESOC '12)*, Lugano, Switzerland, Mar. 2012, pp. 333–339.

[3] A. Bielenberg, L. Helm, A. Gentilucci, and D. Stefanescu, "The growth of Diaspora - A decentralized online social network in the wild," in *Proceedings of IEEE INFOCOM Workshops 2012*. IEEE, Mar. 2012, pp. 13–18.

[4] M. Florian, F. Hartmann, and I. Baumgart, "A Socio- And Locality-Aware Overlay for User-Centric Networking," in *Proceedings of the International Conference on Computing, Networking and Communications (ICNC 2014)*, Honolulu, Hawaii, USA, Feb. 2014.

[5] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta, "PeerSoN: P2P Social Networking — Early Experiences and Insights," in *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems (SNS '09)*, Nuremberg, Germany, Apr. 2009, pp. 46–52.

[6] F. Hartmann and I. Baumgart, "Towards Socio- and Resource-Aware Data Replication in User-Centric Networking," in *Proceedings of the 1st KuVS Workshop on Anticipatory Networks, pp. 20-24, Stuttgart, Germany, September 2014*, Stuttgart, Germany, Sep. 2014.

[7] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB – The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2010.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://portal.acm.org/citation.cfm?doid=362686.362692

[9] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007*, Anchorage, AK, USA, 2007, pp. 79–84.

[10] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks." *Nature*, vol. 393, no. 6684, pp. 440–442, Jun. 1998.

[11] D. Avrahami and S. E. Hudson, "Communication Characteristics of Instant Messaging: Effects and Predictions of Interpersonal Relationships," in *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work*, Banff, Alberta, Canada, Nov. 2006, pp. 505–514.

[12] S. Finster and I. Baumgart, "SMART-ER: Peer-based privacy for smart metering," in *IEEE INFOCOM Workshop on Communications and Control for Smart Energy Systems*. Toronto, Canada: IEEE, Apr. 2014, pp. 642–647.

[13] L. A. Cutillo, R. Molva, and T. Strufe, "Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust," *IEEE Communications Magazine*, vol. 47, no. 12, pp. 94–101, Dec. 2009.

[14] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin, "Persona: An Online Social Network with User-Defined Privacy," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, Barcelona, Spain, Aug. 2009, pp. 135–146.

[15] B. C. Popescu, B. Crispo, and A. S. Tanenbaum, "Safe and Private Data Sharing with Turtle: Friends Team-Up and Beat the System," in *Proceedings of the 12th International Workshop on Security Protocols*, Cambridge, UK, Apr. 2004.