User-Centric Networking: Privacy- and Resource-Awareness in User-to-User Communication

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

Dissertation
von

Dipl.-Inform. Fabian Hartmann
aus Salzgitter

Tag der mündlichen Prüfung:	15. Juli 2016
Erste Gutachterin:	Prof. Dr. Martina Zitterbart
	Karlsruher Institut für Technologie (KIT)
Zweiter Gutachter:	Prof. Dr. Thorsten Strufe
	Technische Universität Dresden

Zusammenfassung

Nutzer-zu-Nutzer-Kommunikation innerhalb einer geschlossenen Gruppe von Endnutzern ist einer der wichtigsten Anwendungsfälle des Internets: Sowohl private, als auch geschäftliche Kommunikationskanäle verwenden Dienste wie Instant Messaging, VoIP-Telefonie und File-Sharing-Angebote, über die Inhalte jeglicher Art ausgetauscht werden. Üblicherweise werden diese Dienste heutzutage über zentralisierte, schnell und zuverlässig verfügbare Cloud-Server von Drittanbietern verfügbar gemacht. Dass Dritte somit am Kommunikationsprozess beteiligt werden, wirft aus Sicht der Nutzer privatsphärenseitige Fragen auf, wird jedoch häufig akzeptiert in Hinblick auf die zuverlässige Bereitstellung dieser Dienste: Die Anbieter erleichtern die Datenspeicherung, wie -übertragung über die persönlichen Geräte der Mitglieder einer geschlossenen Gruppe und tragen so zu einer hohen Datenverfügbarkeit jedes veröffentlichten Datenobjekts bei.

Sofern diese Datenobjekte allerdings lediglich den Mitgliedern der geschlossenen Gruppe zugänglich sein sollen, müssen die Drittanbieter Zugangskontrollen durchführen. Dazu benötigen sie *explizite Anwendungsmetadaten*, um die entsprechenden Nutzer zu identifizieren und zu adressieren. Üblicherweise ist ein eindeutiger Identifikator eines Nutzers auch Teil der expliziten Metadaten. Beispiele hierfür sind E-Mail-Adressen oder Online-Profile auf Social-Media-Plattformen. Sobald Drittanbieter Nutzer über einen eindeutigen Identifikator adressieren, kann die Privatsphäre der Nutzer angegriffen werden, da die Drittanbieter das Nutzungsverhalten der Nutzer somit nachverfolgen können. Durch zusätzliche *implizite Anwendungsmetadaten* erfahren die Drittanbieter beispielsweise durch die Anzahl an Interaktionen, wer einen engen sozialen Kontakt zu dem entsprechenden Nutzer hält, oder über das Datenkonsumverhalten des Nutzers, welche Interessen er oder sie hat. Abhängig von den Merkmalen des Identifikators können sogar Rückschlüsse auf die Identität des Nutzers im realen Leben gezogen werden. Zwar existieren einige Verfahren zum besseren Schutz der Privatsphäre, die die Anonymität der Nutzer gewährleisten, jedoch werden diese üblicherweise dazu genutzt, Verbindungsmetadaten innerhalb des Netzwerks zu verschleiern.

Aus diesem Grund untersucht diese Dissertation **User-Centric-Networking** als eine Alternative zu Drittanbietern in der Nutzer-zu-Nutzer-Kommunikation. User-Centric-Networking basiert auf dem Prinzip der **Selbstversorgung**, d.h. die Übertragung von Datenobjekten geschieht ausschließlich mit Hilfe der persönlichen Geräte der Mitglieder einer geschlossenen Gruppe. Der Vorteil dieses Systems besteht darin, dass die Privatsphäre der Gruppenmitglieder wesentlich besser geschützt werden kann, da keine Anwendungsmetadaten an Dritte übermittelt werden. Bisherige Untersuchungen widmeten sich bereits der Frage, wie Datenobjekte auf ausschließlich vertrauenswürdigen Geräten gespeichert werden können. Dabei nahmen diese jedoch stets ein grundlegendes Vertrauen in einzelne Nutzer an, wie z.B. Freunde oder Familie. Vertrauenswürdigkeit innerhalb von geschlossenen Gruppen, die auf einer Pro-Datenobjekt-Basis stets anders zusammen gesetzt sind, wurde bislang nicht berücksichtigt. User-Centric-Networking stellt somit eine neue Lösung für dieses Problem dar.

Die exklusive Nutzung persönlicher Geräte birgt Nachteile, da diese gemeinhin heterogen im Hinblick auf Verfügbarkeit und Leistungsfähigkeit sind. User-Centric-Networking begegnet der Geräteheterogenität mit Hilfe zweier Eigenschaften: erstens, **Partitionstoleranz** zur Überwindung von zwischenzeitlichen Gerät-Nichtverfügbarkeiten, um Datenobjektübertragungen nachträglich durchzuführen, und zweitens, **Ressourcenbewusstsein**, um die Datenübertragung im Kontext der individuellen Verfügbarkeit und Leistungsfähigkeit der Geräte zu steuern. In diesem Zusammenhang wurden drei sich gegenseitig ausschließende Ziele identifiziert: niedrige Übertragungsverzögerungen, niedrige Kosten und hoher Schutz der Privatsphäre. Abhängig von der Verteilung von Geräten und der individuellen Privatsphäreanliegen innerhalb der geschlossenen Gruppe, kann eine Vorgehensweise zur Datenübertragung vorteilhafter sein, als eine andere. Die technische Lösung dieses Problems bedarf hoher Flexibilität.

Zentrale Beiträge dieser Dissertation sind daher zum einen der Konzeptentwurf des neuartigen User-Centric Networking und seiner drei Entwurfsziele (Selbstversorgung, Partitionstoleranz und Ressourcenbewusstsein), sowie die technische Realisierung dessen. Die technische Realisierung besteht aus drei Teilen, die ineinander greifen: **SODESSON**, **SocioPath** und den sogenannten **Decision Engines**.

SODESSON ist eine Middleware für Anwendungen der Nutzer-zu-Nutzer-Kommunikation. Diese Middleware läuft auf den persönlichen Geräten der Nutzer. Die Anwendungen adressieren über eine Schnittstelle lediglich die Topics, deren Zugriffsrechte von einem speziellen Mitglied der geschlossenen Gruppen – dem Topic Owner – zuvor definiert wurden. Diese Zugriffsrechte werden über eine einheitliche Kontaktliste gesetzt. Zusätzlich unterstützt die Schnittstelle die Trennung von Benachrichtigungen über neue Datenobjekte und den Datenobjekten selbst. Die Kommunikation zwischen Geräten wird von SODESSON selbst nicht organisiert. Stattdessen wird SODESSON durch ein Datenverteilungsprotokoll (Data Distribution Protocol, DDP) ergänzt, welches diese Aufgabe übernimmt.

SocioPath ist ein DDP für SODESSON, welches das Konzept des User-Centric Networking technisch realisiert. Die Kerneigenschaften von SocioPath sind die Umsetzung der drei Entwurfsziele des User-Centric Networking: Selbstversorgung, Partitionstoleranz und Ressourcenbewusstsein. Als Basis für die Selbstversorgung unterhält jedes Gerät eine Liste aller Geräte aller Nutzerkontakte und kann somit direkt mit ihnen kommunizieren. Jeder Nutzer kann die Rolle eines Topic Owners einnehmen und seinen Kontakten Zugriffsrechte für das jeweilige Topic zuweisen. Benachrichtigungen über Datenobjekte werden dann über die Geräte des Topic Owners geleitet und von diesen an die Empfänger weitergeleitet. SocioPath setzt *Partitionstoleranz* über sogenannte Zustandsreparaturen um. Eine Zustandsreparatur ist ein Prozess, bei dem zwei Geräte Informationen zu existierenden Datenobjekten austauschen und somit fehlende Datenobjekte erkannt werden. Dies geschieht üblicherweise nachdem mindestens ein Gerät zeitweise nicht für andere Geräte verfügbar war. Zur Erkennung der fehlenden Benachrichtigungen werden diese in eine speichereffiziente Bloom Filter-Datenstruktur eingefügt. Zur weiteren Reduzierung des Speicheraufwands merken sich Geräte paarweise den Zeitpunkt der letzten erfolgreichen Zustandsreparatur.

SocioPath wurde entworfen, um verschiedene Weiterleitungsstrategien für Datenobjekte zu unterstützen, da abhängig von den verfügbaren Ressourcen eine Strategie besser sein kann als eine andere. Dadurch ist es SocioPath möglich, *Ressourcenbewusstsein* zu schaffen. Umgesetzt wird jeweils eine Strategie als Decision Engine, welche das Verhalten von SocioPath an zuvor offen gehaltenen Punkten im Protokollablauf fest definiert. Um dieses Konzept zu belegen, wurden die drei verschiedenen Decision Engines *Instant-to-All*, *Offload-First* und *Helping-Friends* entworfen, die sich auf jeweils unterschiedliche Ziele fokussieren: niedrige Verzögerungen, Kostensparsamkeit oder hohe Privatsphäre innerhalb der geschlossenen Gruppe.

Das technische Konzept – bestehend aus SODESSON, SocioPath und *Decision Engines* – wurde als Prototyp für den Overlay-Netzwerk-Simulator *OverSim* implementiert. In einer umfangreichen Evaluierung von zwei Anwendungsszenarien und verschiedenen Geräteverteilungen hinsichtlich Zustellverzögerungen und Kosten wurde belegt, dass durch die Kombination von direkten Zustellungen und Zustandsreparaturen Datenobjekte mit Verzögerungen zugestellt wurden, die nahe (< 1% Differenz) an einer theoretischen Untergrenze lagen. Durch die Verwendung von Offload-First statt Instant-to-All konnten die Maximalkosten für ein Gerät bis zu 79,6% gesenkt werden. Gleichzeitig wurde jedoch die Problematik der zentralen Aufgabe des Topic Owners sowohl durch hohe Kosten als auch Verfügbarkeitsflaschenhals belegt.

Während also die grundlegende Funktionsweise der Idee des User-Centric Networking durch simulative Modelle belegt wurde, stellt die Entlastung des Topic Owners ein wichtiges Thema für zukünftige Forschungsarbeiten dar.

Contents

List of Figures

Chapter 1

Introduction

The rise of everyday Internet usage has addressed different major human needs. One of these is private *user-to-user communication in a closed group* (**U2U communication**): both personal and business communication processes make use of services such as instant messaging, VoIP telephony and sharing files with content types of all kinds from a spreadsheet to high definition videos. Due to the strong demand for such services, companies have started to offer specific services, such as Online Social Networks [33], live collaboration [44], telephony [103] or file exchange and synchronization platforms [29]. Typically, providers for such services are *Centralized Service Providers (CSPs)*. A CSP is a *third party* that hosts its service(s) on highly available cloud web servers which are completely under the CSP's own administration. This approach enables a CSP to bring convenient advantages to its customers, i.e. the users: ubiquitous access for and synchronization across different personal devices as long as Internet access is available to the personal devices. Additionally, this design has a lower complexity compared to other approaches: for example, user data is addressed, storage and retrieved within a logically single storage, which simplifies data management for the CSP.

Involvement of a third party in private communication may be undesirable by the users, especially if the communication process is confidential. However, a third party is a necessary means to service provision in a centralized networking paradigm. A common solution for the customer to ensure content data confidentiality is end-to-end encryption. It prevents a third party to read the content data two users exchange. Furthermore, connection metadata such as IP addresses can be obfuscated by using proxies or an onion routing service such as *Tor* [25]. However, the users still have to give away metadata on the application layer to the CSP, i.e. *application metadata*.

Application metadata includes two types of information: firstly, *explicit metadata* is required for the service and to correctly establish communication processes. To this end, users have typically a unique identifier so that they can be addressed by other users and by the CSP. These identifiers are used to enable access control and deliver data objects to the correct users. Examples for explicit metadata are email addresses, online social network profiles or entries in an address book that is synchronized with a CSP. Trivially, if the CSP requires this metadata to establish a communication process between user A and B, the CSP learns that user A and user B have some type of social connection. The fact that the CSP has this knowledge already affects both *A*'s and *B*'s privacy – even if the data objects are encrypted and anonymization is ensured on a lower layer (e.g. via Tor). As a remedy, *A* and *B* could pursue *unlinkability* by using multiple, short-lived identities. However, this results in additional effort for the users and creating additional identities is often not feasible, depending on the service: for example, WhatsApp [116] uses each user's cellphone number for identifying and addressing him. It is not possible for a user to generate an arbitrary valid cellphone number. Another alternative is the use of broadcast schemes, making identities not required anymore. This approach raises scalability issues for increasing numbers of users in the service.

Secondly, *implicit metadata* accumulates at the CSP, given that each user has a unique, long-lived identity. Examples are the date and time of a sent email or the frequency of visiting a social network profile. Over time, the CSP 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), the CSP might infer even more information about a user.

Decentralization is one approach to alleviate these implications. Here, a service's users' content data and metadata is not accumulated at one provider, but distributed across different providers under different authoritative administrations. Therefore, it is not possible for a single provider to gain complete information with regard to a given service. However, depending on the architecture of the decentralized network, content data and application metadata might still accumulate on one specific provider's end, e.g. due to high popularity. Two common provider schemes for decentralized U2U communication are the following.

- *Federated service providers (FSPs)*: This approach is similar to classic email exchange. Users are free to select their own FSP to store data on a highly available server. A user's identifier consists of an FSP-widely unique username and a globally unique FSP identifier. Since two providers interoperate via a server-to-server protocol, it is possible for a user at FSP *X* to address a user at FSP *Y* and vice-versa. Other FSPs, such as provider *Z*, do not have to be involved in this process. The amount of user data visible to a specific FSP depends on the distribution of users to FSPs. Possible influences for skewed distributions are different FSPs' popularities, usage charges or geographic locations. If the majority of the overall data amount accumulates at one provider, a situation similar to a CSP arises. At any rate, an FSP has a complete view on all its own users.

- *Structured P2P overlay networks*: This approach is based on *Distributed Hash Tables* (DHT) and is a common solution for a distributed data storage without a centralized organizer instance. All devices that access the storage, i.e. are able to read from and write to it, share the overall load of stored data. Hence, each participating device is a small server in regard to a specific subset of data. The participating devices form an *overlay network* together. An overlay network is a virtual network which uses an *underlay network* – such as the Internet – as a substrate for communication between two devices. While not users per se are identified here, each user device in the overlay network has a unique *NodeID*. Read / write requests regarding a specific data object are delegated to a responsible device.

> To determine responsibility, a key is created for each data object. The key is mapped to the same number space as the NodeIDs by a hash function, rendering the closest device responsible in terms of a well-defined metric. Given that both NodeIDs and key hashes are evenly distributed, the number of responsibilities is fairly balanced between the participating devices. In order to tackle device unavailability, data objects also get replicated to devices with the next-to-closest NodeIDs. This means that a user has neither influence nor clear insight which devices store his data and who controls the responsible devices. If the DHT protocol offers no protection against Sybil attacks [28], it is even possible for an attacker to control complete parts of the key/NodeID number space.

Therefore, both types of decentralized provider schemes expose application metadata to third parties. This affects the privacy of a closed group negatively, although the global view of a CSP is prevented here.

This thesis presents and evaluates **User-Centric Networking** as a possibility to increase the privacy of the members in a closed group. User-Centric Networking features so-called *self-sufficient* U2U communication, where third party application providers are not involved in the communication of a closed group. Hence, application metadata does not get exposed to third parties and the closed group members' privacy is improved.

1.1 Overview

The involvement of third-party providers in U2U communication is an inherent privacy problem. This section discusses this problem on an overview level. First, a basic scenario for U2U communication – including a closed group and one or multiple third-party providers – is described. The assumptions made in this scenario are the basis for all considerations throughout this thesis. Second, privacy considerations based on this scenario are made.

1.1.1 Basic Scenario

While different cases of U2U communication are conceivable (identifiable / anonymous sender, well-known / unknown recipient group, etc.), this thesis assumes the following scenario: a U2U communication process equals the sharing of a **data object** among the members of a closed group. A data object conveys confidential information (only meant for the eyes of the group members) and can be passed from one actor to another – be it a group member or a third party provider. The creator of the data object is assumed to be not anonymous to the recipients. Instead, a **linkability** from the data object to the creator exists: the creator's identity is inherently tied to the data object, which gives the data object additional information. For example, a message with the content "Let's meet for dinner!" is only useful to the reader when the creator is known to him. Other conceivable cases for U2U communication, e.g. with an anonymous sender, shall not be further regarded here.

The scenario follows the notions of topic-based publish/subscribe communication. Here, each data object is associated to a **topic**. A topic is a specific semantic context, which is used for grouping data objects with the same audience. For example, a number of photos are different data objects. These photos can be grouped into a single photo album which is to be shared with the same users. In this case, the different data objects have the same topic.

Each topic has exactly one **topic owner**. The topic owner is one specific user that controls *access rights* for the topic: He defines for the given topic which other users are **allowed publishers** and which users are **allowed subscribers**. Allowed publishers are allowed to publish data objects, i.e. to become a publisher of a data object with the given topic. Only allowed subscribers have the option to indicate their interest in the given topic and become actual **subscribers**. Furthermore, the topic owner is able to add new and remove existing allowed publishers and allowed subscribers.

The sharing of a data object δ_t with topic t is framed by a first and a final step: First, one allowed publisher for topic t creates and *publishes* δ_t – this allowed publisher for topic t becomes the **publisher** of δ_t. Finally, δ_t gets *delivered* to the subscribers of topic t. The subscriber can now *consume* δ_t. All data objects with the same topic t are meant to be delivered to the same set of users, i.e. the subscribers of t.

Delivering a data object from the publisher to the subscribers requires additional steps in-between. During these steps, third party provider(s) are involved as additional actors. Depending on whether a centralized or decentralized provider scheme is used, there are one or multiple third party providers involved. Furthermore, the user devices may be in separate physical networks under different administrations.

Therefore, in this basic U2U communication scenario, two types of actors can be identified, which will be discussed in the following:

- Members of the closed group for a data object δ_t
- Third party provider(s)

This basic scenario is also depicted in Figure 1.1.

Members of the closed group

A closed group is associated to one specific data object δ_t with topic t. The members of the closed group assume one or multiple of the following roles:

- Topic owner of t
- Publisher of δ_t
- Subscribers of t

Note that this closed group is defined on a per-data-object basis, not a per-topic basis. A closed group only includes those users that are actually meant to access the respective data object – here: δ_t. The group does not include any other allowed publishers or allowed subscribers. Therefore, another data object ϵ_t from another (allowed) publisher for topic t is associated to another closed group, consisting of the topic owner of t, the publisher of ϵ_t and the subscribers of t.

The closed group for a data object δ_t is displayed in the blue background of Figure 1.1.

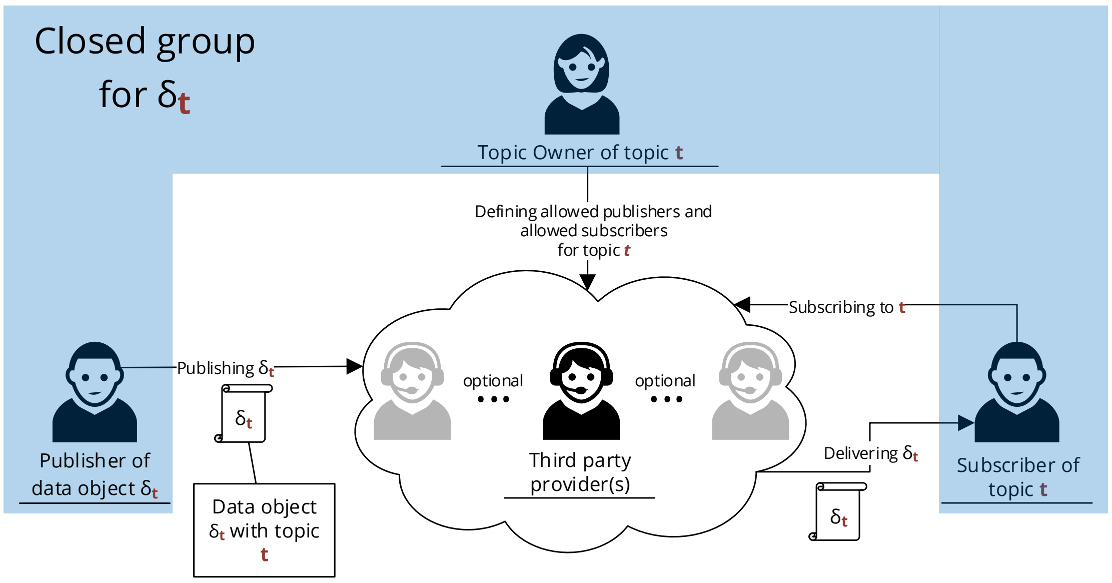


Figure 1.1: *Basic scenario of user-to-user communication*

Third party provider(s)

For the sake of simplicity, a single CSP is assumed here. The following considerations also apply if multiple third-party providers are involved, i.e. a decentralized provider scheme is chosen.

The CSP has two tasks with regard to a data object δ_t. First, it has to adhere to the access rights as defined by the topic owner, i.e. it must verify that δ_t was created by an allowed publisher and it must deliver δ_t to no other user than the subscribers (*access control*). Second, it has to deliver δ_t to all subscribers of t (*data delivery*).

Generally, access control can be defined in two ways: first, each user can have his unique user identifier and his associated rights can be defined by a tuple {user identifier, rights}. Second, both the users and the data object may have one or multiple attributes assigned. Only if a user's attributes and the data object's attributes overlap, the user has the access rights that is bound to a common attributes. One example for attribute-based access control is given [56]: "*all Nurse Practitioners in the Cardiology Department can View the Medical Records of Heart Patients*".

While attribute-based access control is suitable for a potentially unlimited user group that is not further specified (besides their attributes), the regarded U2U communication scenario is different. The topic owner selects the allowed publishers and allowed subscribers from a limited pool of users known to him, i.e. his contacts. He makes his selection made via user identifiers. Thus, in order to enforce access only for the closed group, the user identifiers are revealed to the CSP. First, when the topic owner of topic t defines the access rights, he passes a list of allowed publishers and allowed subscribers to the CSP. This list must consist of unique user IDs. Whenever a user interacts with the CSP in order to publish a data object δ_t, this user has to pass his user ID to the CSP. In turn, the CSP refers to the list of allowed publishers and checks if this user is on it. The same concept applies to a user that tries to subscribe to t at the CSP: the CSP checks whether this user is an allowed subscriber. Second, in order to deliver δ_t to all subscribers, the CSP needs to refer to the list of actual subscribers of t and deliver δ_t to these and only these.

Therefore, the CSP requires the users' identification to fully comply to the group members' expectations: to deliver δ_t inside the closed group after δ_t was published by an allowed publisher.

1.1.2 Privacy Considerations

By gaining knowledge about the involved users' identities, a third-party provider gains information that goes beyond the access control and data delivery tasks: the provider learns that there is a social relationship between the publisher and the topic owner, as well as the subscribers and the topic owner. This is an obvious conclusion, since the topic owner defined access rights for the corresponding users with regard to data objects that are not public, but meant to be shared inside the closed group only.

The provider might also infer relationships between the publisher and the subscribers, depending on the inter-publishing times and publishers of data objects with the same topic. For example, if two allowed publishers publish alternately, this might indicate a dialog where messages are sent back and forth.

Note that the provider does not even need to be able to read the *content data* of the data object (e.g. due to encryption) to draw these conclusions. Furthermore, it is also possible to draw conclusions on the content data by regarding metadata – here: the identities of the communicating parties – only. The Electronic Frontier Foundation (EFF) presents some graphic examples in [32]:

> - "*They know you called the suicide prevention hotline from the Golden Gate Bridge. But the topic of the call remains a secret.*"
> - "*They know you got an email from an HIV testing service, then called your doctor, then visited an HIV support group website in the same hour. But they don't know what was in the email or what you talked about on the phone.*"
> - "*They know you called a gynecologist, spoke for a half hour, and then searched online for the local abortion clinic's number later that day. But nobody knows what you spoke about.*"

Therefore, each identifiable member of a closed group needs to **trust** each involved third-party provider to respect the member's privacy, since their privacy cannot be secured in the U2U communication scenario.

There exist privacy enhancing techniques (PETs) that can achieve third-party anonymity, i.e. keep the identity of publisher or subscriber hidden from a given third-party and thus hide their connection. Usually, these are used for obfuscating connection metadata. For example, *Tor* [25] inserts intermediary hops between source and destination. The route via these hops is defined by the source, which chooses intermediary nodes by its own from a large pool of candidate nodes. To enable such a system for application metadata, however, a large number of application providers must be willing to participate in such a system, offer themselves as intermediary hops and make themselves available to the publisher for choice. Participating in and maintaining such an extra system results in a larger effort for the third-party providers, as well as the publisher. Generally, the motivation for a third-party provider in enabling application metadata anonymity is questionable, especially if its business model is built on analyzing user behavior, user interests, etc.

As an alternative to these considerations, third-party providers could be removed altogether, which is the idea pursued in User-Centric Networking.

1.2 Problem Statement

Based on the considerations above, the following problem statement is made:

U2U communication is about sharing a data object within a closed group. This data object is only meant for the eyes of the group members. Unlike other scenarios, e.g. file sharing, the publisher is not anonymous to the subscribers. Instead, there is an inherent association between the data object's content and its publisher's identity. Additionally, due to the data object's confidentiality, the subscribers must be identifiable in order to discern them from users that must not access the data object.

Due to the identifiability of publisher and subscribers, the closed group members' privacy is threatened as soon as third-party providers are involved in access control and data object delivery: the closed group member's identities and their communication patterns get exposed to these providers via application metadata.

While there exist privacy enhancing techniques (PETs) which aim for third-party anonymity, these are hardly applicable for application metadata. For example, there exist browser plugins that end-to-end encrypt data objects before uploading them to Facebook (e.g. [50], [70]). However, the users still except that the data objects get delivered to the correct users. Thus, Facebook still needs to make a connection between users, regardless of the encrypted data content. A system which enable application metadata privacy would require the cooperation of application metadata providers and result in larger effort for both the users and the third-party providers. It is unlikely that a third-party provider has an interest in implementing PETs for application metadata.

Since third-party providers present themselves as obstacles for the users in these consideration, the idea is to simply remove third-party providers altogether. This thesis investigates User-Centric Networking as an approach to increase the privacy of closed group members in U2U communication. User-Centric Networking removes third-party providers from the tasks of access control and data object delivery and gives these tasks to the group members themselves.

The scope of User-Centric Networking is to enable the sharing of data objects of all kinds in relatively small groups (up to 100 users). Such data objects may range from an instant message to a file of multiple gigabytes. User-Centric Networking is application-agnostic: both private data objects (photos, videos, ...) as well as enterprise data objects (presentation slides, spread sheets, ...) are possible.

1.3 User-Centric Networking

User-Centric Networking (first presented and refined in publications [6] [38] [51] [52])[1] is a provider scheme for U2U communication which aims at increasing the group members' privacy. Here, third-party providers are not involved in the communication of a closed group at all. Instead, access control and data object delivery are performed by the group members themselves. As a result, application metadata does not get exposed to third parties and the closed group members' privacy is improved.

The group members still have to trust each other here: each group member must keep the data object confidential. Also, each group member must not expose any known group members' identities and therefore respect their privacy. It is argued that if one group member has access to the data object's confidential contents or the identitity of other group members is revealed to him in the first place, he could give it away regardless of any provider scheme. Detection of such a behavior is outside of this thesis' scope.

Each closed group is associated to a specific data object – general trust in other users outside of the closed group, such as friends, family or work colleagues, is not required here.

User-Centric Networking assumes that modern personal devices controlled by the closed group members, are powerful enough to be provider devices, i.e. deliver data objects and handle access control by themselves. Therefore, the devices of a closed group's members are not only used to publish / consume a data object, but also act as provider devices for storing and delivering data objects to subscriber devices. All types of devices that are at a user's disposal can be leveraged here – ranging from embedded systems over smartphones up to PCs and small personal servers, e.g. network-attached storages. However, personal devices are heterogenous in terms of resourcefulness and availability. User-Centric Networking aims to take this heterogeneity into account.

User-Centric Networking has the following three main targets:

- **Increased application metadata privacy by self-sufficiency**: All storage of data objects for a closed group is performed by the group members' personal devices. Access control and data delivery between publisher and subscribers are performed by devices controlled by users of the respective closed group only. Third-party providers are completely excluded from this approach.

  Only users that are meant to access a specific data object are involved in the delivery of said data object from publisher to subscribers – no third parties, even if they are otherwise known and trusted friends of any group member.

  This type of U2U communication is called *self-sufficient U2U communication* or **self-sufficiency**. By completely removing third-party providers from U2U communication, neither content data nor application metadata is visible to third parties anymore.

- **Partition tolerance**: Data delivery in User-Centric Networking relies on personal devices. Personal devices can lack Internet connectivity at times, be moved from one local network

[1] Besides the definition used here, the term "User-Centric Networking" is overloaded by different definitions throughout the literature. See Section 3.7.3 for details.

to the other or perform opportunistic networking via ad-hoc or delay-tolerant networks with other devices. Such diverse connectivity situations can result in disjunct partitions, where some devices are able to communicate with each other while other devices are not available.

Devices within the same partition should be able to exchange data objects, regardless of any devices of the same or other users outside that partition. For example, this would enable U2U communication for two users on an airplane, without any Internet connectivity, but with their local devices communicating directly via Bluetooth or WiFi Direct. This local ad-hoc network forms a partition on its own.

Data objects should also be transferable from one partition to the other. If two partitions merge (e.g. a disrupted link between two network becomes functional again), the previously published data objects can now be delivered to the remaining subscriber devices. Another possibility is a mobile topic owner device which carries a data object from network $\mathcal{A}$ to network $\mathcal{B}$, comparable to a delay-tolerant pocket switched network.

- **Resource-aware leverage of personal devices**: As mentioned above, personal devices are heterogenous in terms of resourcefulness and availability. User-Centric Networking takes these differences into account when coordinating access control and data delivery across the devices. As a result, the combined resources of all group members' devices can and should be used as provider devices. If possible, provider tasks shall mainly be performed by devices with higher availability and capabilities, such as storage, fast and cheap network connectivity, etc.

  The sum of all devices controlled by the members of a closed group is the closed group's *device pool*. The combined resources of the device pool can and should be used for provider tasks, regardless on which specific device each user publishes data objects or consumes data objects of subscribed topics.

1.4 Building Blocks for User-Centric Networking

The core of this thesis are two major building blocks for enabling User-Centric Networking on personal devices: First, SODESSON – a generic service for U2U communication which runs as a middleware on each personal device and accepts data objects from applications. Second, SocioPath – a self-sufficient Data Distribution Protocol (DDP) which complements SODESSON by handling the communication. SocioPath is both partition-tolerant and resource-aware and therefore fulfills the targets of User-Centric Networking.

1.4.1 SODESSON Middleware

SODESSON provides a topic-based publish/subscribe service to applications running on the same personal device. With a small set of methods, an application can publish data objects for topics or subscribe to topics that were previously set up by a topic owner. SODESSON makes a best effort to deliver that data object to the devices running the subscribed applications.

SODESSON is not only suitable for User-Centric Networking, but is instead a generic middleware for U2U communication. It handles applications, data objects and contacts on a topic/user-addressed level and is agnostic of any devices. It needs to be complemented with a Data Distribution Protocol (DDP) which handles all inter-device communication.

SODESSON fulfills the following requirements:

- **Application interface:** The publish/subscribe service needs to offer a unified interface for the applications to send their data objects to and receive data objects from the network. Multiple applications may run at the same time on one device, which also requires a multiplexing mechanism. Additionally, the interface shall allow a topic owner to maintain the allowed publishers/allowed subscribers for each topic he owns. Likewise, users shall be able to subscribe to topics via this interface.
- **Abstraction from devices:** One aspect of User-Centric Networking is the assumption that each user can have multiple devices. An application should not have to care about addressing the "correct" device (i.e. the one which a user controls to generate and consume application data), but instead have an abstraction on the user level. In SODESSON, an application only addresses topics. Devices are never addressed by applications.
- **Contact management:** A topic owner selects the allowed publishers and allowed subscribers for a topic from a specific set of users. In SODESSON, this specific set of users is the topic owner's *contact list*. This contact list should be independent from the applications and consistent across all devices of the same user. Adding other users to the contact list requires a procedure where two users can agree on being mutual contacts, including exchanging required information for addressing each other.

1.4.2 SocioPath

SocioPath is a self-sufficient Data Distribution Protocol (DDP) which complements SODESSON. It handles all inter-device communication, i.e. accepting data objects from a publishing device and delivering data objects to subscriber devices. Additionally, SocioPath verifies the respective access rights, i.e. ensures that only data objects from allowed publishers get delivered and that only subscription requests from allowed subscribers are accepted.

SocioPath tackles the three main targets of User-Centric Networking – self-sufficiency, partition tolerance and resource awareness – as follows:

Self-sufficiency

In SocioPath, only the closed group's device pool are provider devices for a given data object – there are no third-party providers. The lists of allowed publishers, allowed subscribers and subscribers for a given topic are stored on the topic owner's devices only. Each published data object has to pass a device of the topic owner's device first, since only the topic owner's devices know which devices are subscriber devices. Therefore, the topic owner has full control over defining the allowed publishers and allowed subscribers of the respective topic.

Partition tolerance

SocioPath deals with intermittent unavailabilities of personal devices. As long as two devices are available for each other, they exchange their state at certain times and repair possible inconsistencies. Until the repair, each device works on its own local view and fully participates in SocioPath communication: state inconsistencies are not seen as fatal, but accepted as a regular aspect.

This makes SocioPath tolerant to partitions. For example, if two devices a and b only communicate in a local network (Partition 1), but only a later gains Internet access and can communicate with another device c (Partition 2), c can gain the same state as a and b, even with b not being part of Partition 2.

Resource-awareness

SocioPath has two different approaches to deal with device heterogeneity in User-Centric Networking.

- **Decoupling notifications and data object retrievals**: If SocioPath would select provider devices with low resources and would forward a high number of large data objects to these provider devices in an unsolicited way, this might result in high costs for the user who controls that device. The same applies to a subscriber, if a provider device delivers data objects to a subscriber's device with low resources. In order to prevent such situations and still enable SODESSON's publish/subscribe service, SocioPath decouples into two steps: published data objects which exceed a certain size are at first indicated by a small *notification* with metadata about the object (topic, size, etc.). Such a notification is delivered to each subscriber device. Based on the metadata in this notification, either the user or the device itself (see below: Decision Engines) can decide whether to *retrieve* the actual data object.

- **Exchangeable Decision Engines**: There are multiple possibilities how notifications and data objects are forwarded inside a given device pool. Depending on the device distribution in regard to resources, availability and the number of devices per publisher, topic owner and subscribers, one strategy might be more preferable than the other. Therefore, the SocioPath is not a "hardwired" protocol, but instead defines a set of specific events – from publishing a data object via access control by the topic owner devices to notifying subscriber devices about / letting subscriber devices retrieve a data object. SocioPath is then complemented by a Decision Engine which implements a strategy for handling these events. Different Decision Engines handle events differently and therefore influence SocioPath's behavior.

1.4.3 Trade-Offs in Designing Decision Engines

When the event handling strategy for a Decision Engine in SocioPath is defined, there can be different preferable goals, which unfortunately are mutually exclusive.

For example, greedy automatic retrieval of even large data objects results in short access time for the subscriber, if it is already downloaded to the device by the time the subscriber wishes to consume the data object on that device. However, such a strategy may result in high resource costs, e.g. when downloading gigabytes of data objects via a metered connection. On the other hand, lazy retrieval which always has to be user-demanded gives the user full control over the costs, but may result in higher access delays for him.

Therefore, each Decision Engine must find a trade-off between the following three goals:

- Resource conservation
- Low delays
- Group members' privacy

Goal: Resource conservation

User-controlled devices are very heterogenous in terms of computing power and availability. According to [41], "*8 in 10 internet users now have a smartphone and almost half own a tablet.*" A smartphone has at least limited Internet access most of the time, but bandwidths provided by cellular networks fluctuate with user mobility. Additionally, mobile connectivity is often metered, i.e. paid by consumed megabytes. Finally, battery power is limited for mobile devices. Being provider devices – i.e. storing large data objects and delivering them to other devices – can be unacceptable for the users of limited devices, which is why ***resource conservation*** is an important aspect in User-Centric Networking.

Goal: Low delivery delays

In contrast to highly-available cloud servers, user-controlled devices also suffer from frequent temporary unavailability (*churn*) [73]: Smartphones get switched off at night, laptops get sent into sleep mode and mobile reception breaks away when driving through a tunnel. If a data object is only stored on such a device, it is not accessible from other devices during offline times. Having to wait for the storing device to return online, leads to a high delay, i.e. the time a receiver's device spends online until the data object reaches it. High delays can result in an unacceptable user experience and therefore render User-Centric Networking an unhelpful alternative to third-party providers. Hence, ***low delays*** are an important goal in User-Centric Networking.

Goal: Group members' privacy

Application metadata privacy is not only relevant when third-party providers are involved. It is still an issue in User-Centric Networking, even though on a smaller scale, i.e. within a closed group.

Given a closed group of three users – A, B and C. Only A publishes data objects, B and C are subscribers. Assume that B has a very resourceful device which is suitable for storing the published data objects and delivering them to C's device. In this case, B learns that C is also a recipient of A's data objects. Depending on the application use-case, this might not be wanted by A or B[2].

This translates to the following real life analogy: If a confidential piece of information is to be given from a sender to a closed recipient group, there is a difference if the sender speaks to the

[2]Detailed considerations about trust dependencies and possible adversary types here will be made in Chapter 3.

whole group at the same time in a single room or to each recipient one-by-one, urging them to keep the conversation confidential. In the former case, every recipient also knows who the other recipients are. In the latter case, they do not. Depending on the use case, this distinction can be important. Translated to self-sufficiency, the subset of devices that is provider device for other subscribers is key to different levels of privacy.

Therefore, the subset of provider devices within the device pool, i.e. devices that store a data object and deliver it to the subscribers is key to different levels of group members' ***privacy*** in User-Centric Networking.

Mutual exclusiveness of the three goals

- *Privacy vs. resource conservation*: A higher level of privacy is reached if the topic owner sends a given data object it to all subscribers via his own devices only instead of letting subscribers' devices help others with the delivery. This way, the topic owner prevents that these subscribers gain knowledge about which other users are also subscribers. However, some subscribers might have more resourceful devices than the topic owner. In this case, help with the distribution would be desirable for the topic owner in terms of resource conservation. If the topic owner wants to prevent this for privacy reasons, he has to rely on his own device resources.

- *High privacy vs. low delays*: A lower delay means to deliver a data object to as many subscriber devices within the User-Centric Network as soon as possible: if one of these devices becomes unavailable, others can still be provider devices. As in the first case, subscribers may help other subscribers here. However, the user who controls a provider device can learn which other users are subscribers.

- *Low delays vs. resource conservation*: As described in the introductory example of Section 1.4.3, a lower delay can be reached by delivering a data object to as many subscriber devices as soon as possible – regardless of any resource constraints. This includes limited devices for which storing, delivering or receiving large objects can be unacceptable.

Therefore, the provider device selection needs to be carefully weighed with regard to these three mutually exclusive goals, depending on the application use-case – a trade-off needs to be found. For example, for small instant messages, application metadata privacy might be more important than resource conservation.

Decision Engines

This thesis offers three different strategies to deal with the presented trade-off requirements: **Instant to All**, **Offload First** and **Helping Friends**. Each Decision Engine focuses on two of the three goals, as displayed in Table 1.1.

Table 1.1: *Overview on the Decision Engines*

	High privacy	Low delays	Resource conservation
Instant-to-All	✓	✓	✗
Offload-First	✓	✗	✓
Helping-Friends	✗	✓	✓

1.5 Outline

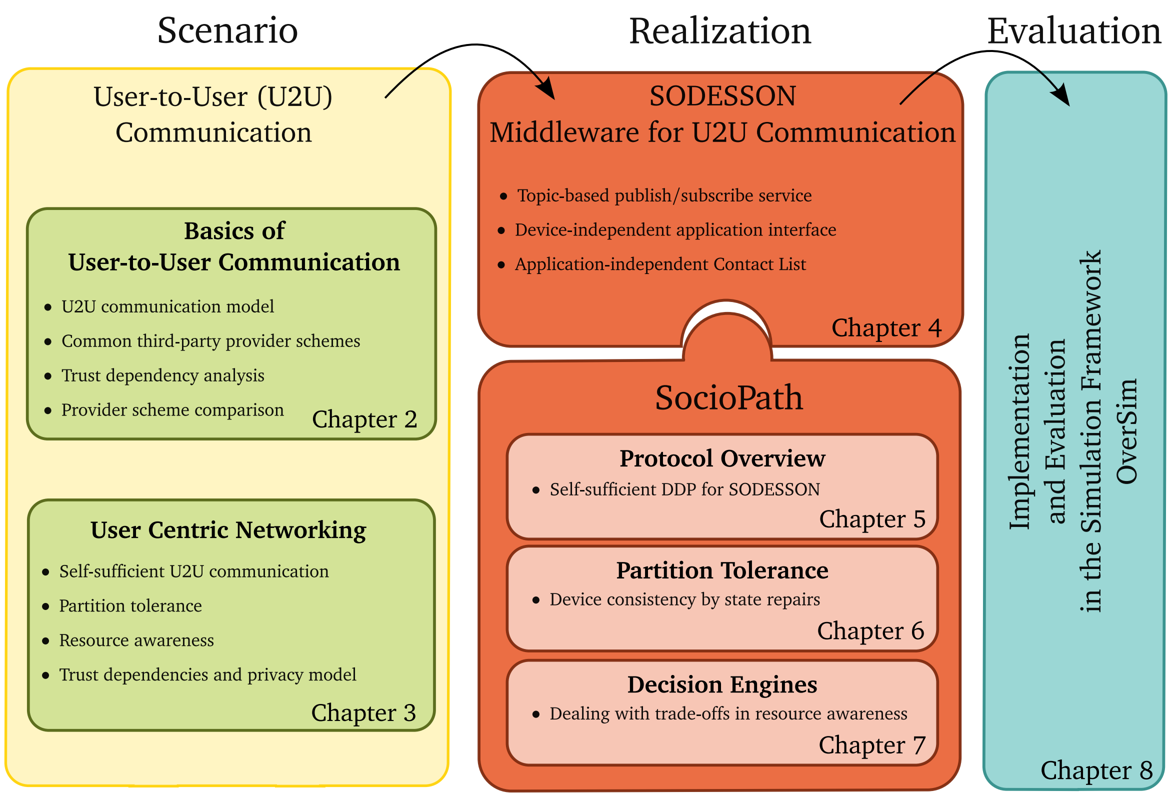
This thesis will be structured as displayed in Figure 1.2: Chapters 2 and 3 will discuss the scenario of U2U communication. As such, Chapter 2 will cover the basics for U2U communication that includes third-party providers. To this end, it will present a model for U2U communication and three common third-party provider schemes. For these provider schemes, a trust dependency analysis will be made. Also, in anticipation of Chapter 3, the existing provider schemes will be compared with regard to design goals in User-Centric Networking.

Chapter 3 presents User-Centric Networking as a new provider scheme concept. It presents a formal definition for a User-Centric Network and discusses the three design goals self-sufficiency, partition tolerance and resource awareness. The trust dependency analysis from Chapter 2 is also done for User-Centric Networking here. Finally, a privacy model based on the previous considerations is presented.

Chapters 4 to 7 will present a concept to technically realize User-Centric Networking. This concept consists of two major parts: first, SODESSON (Chapter 4), a middleware for U2U communication which offers a topic-based publish/subscribe service for U2U applications will be presented. The main component is its application interface which will be explained in detail, including an step-by-step example. The second part is SocioPath, a self-sufficient Data Distribution Protocol for SODESSON. The discussion of SocioPath is split across three chapters, each of them discussing one specific aspect: overview on the self-sufficient protocol (Chapter 5), partition tolerance with the help of state repairs (Chapter 6) and Decision Engines (Chapter 7).

In Chapter 8, this realization of User-Centric Networking is evaluated as an implementation of SODESSON/SocioPath for the overlay simulation framework *OverSim* [7]. Here, two communication scenarios with regard to delivery delays and costs will be evaluated. Two decision engines – Instant-to-All and Offload-First – are compared.

Finally, a conclusion with perspectives on possible future work will close this thesis.



Scenario
Realization
Evaluation
User-to-User (U2U) Communication
Basics of User-to-User Communication
• U2U communication model
• Common third-party provider schemes
• Trust dependency analysis
• Provider scheme comparison
Chapter 2
User Centric Networking
• Self-sufficient U2U communication
• Partition tolerance
• Resource awareness
• Trust dependencies and privacy model
Chapter 3
SODESSON Middleware for U2U Communication
• Topic-based publish/subscribe service
• Device-independent application interface
• Application-independent Contact List
Chapter 4
SocioPath
Protocol Overview
• Self-sufficient DDP for SODESSON
Chapter 5
Partition Tolerance
• Device consistency by state repairs
Chapter 6
Decision Engines
• Dealing with trade-offs in resource awareness
Chapter 7
Implementation and Evaluation in the Simulation Framework OverSim
Chapter 8

Figure 1.2: *Overview on the upcoming chapters*

Chapter 2

Basics of User-to-User Communication

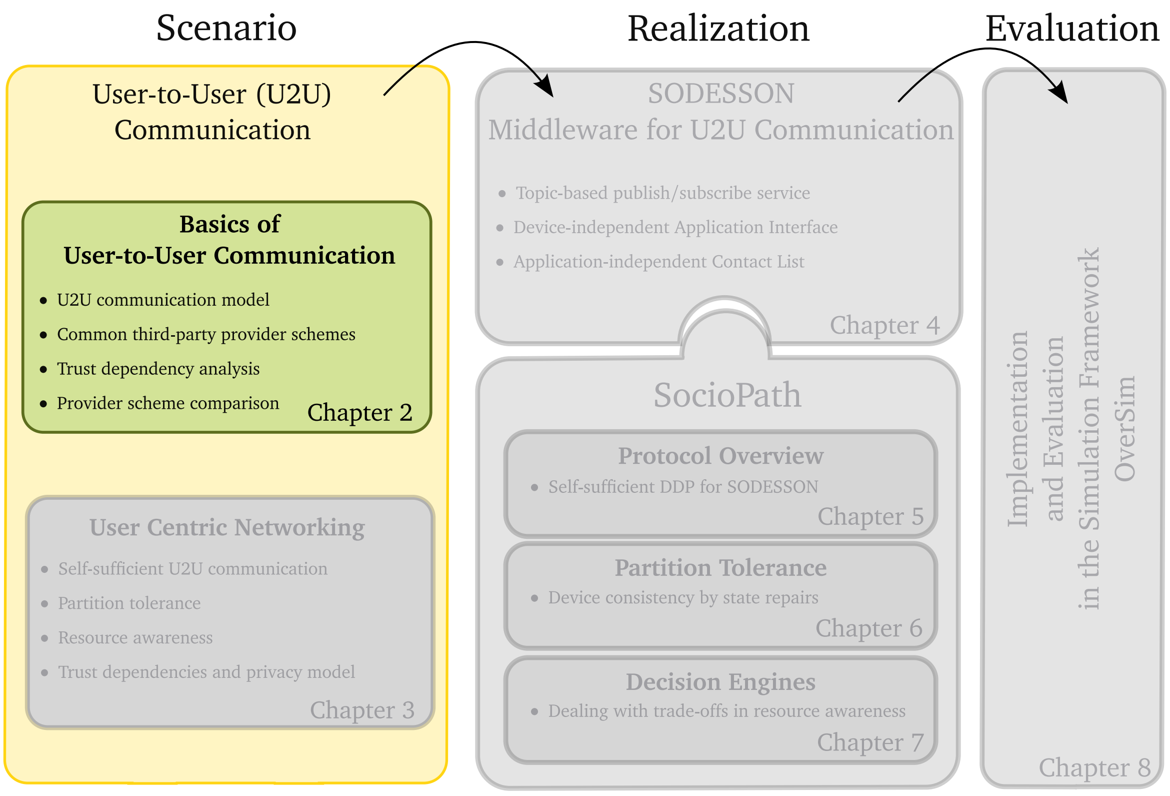

Scenario
Realization
Evaluation
User-to-User (U2U) Communication
Basics of User-to-User Communication
U2U communication model
Common third-party provider schemes
Trust dependency analysis
Provider scheme comparison
Chapter 2
User Centric Networking
Self-sufficient U2U communication
Partition tolerance
Resource awareness
Trust dependencies and privacy model
Chapter 3
SODESSON Middleware for U2U Communication
Topic-based publish/subscribe service
Device-independent Application Interface
Application-independent Contact List
Chapter 4
SocioPath
Protocol Overview
Self-sufficient DDP for SODESSON
Chapter 5
Partition Tolerance
Device consistency by state repairs
Chapter 6
Decision Engines
Dealing with trade-offs in resource awareness
Chapter 7
Implementation and Evaluation in the Simulation Framework OverSim
Chapter 8

Figure 2.1: *Placement of Chapter 2 in the big picture*

This chapter covers the state-of-the-art in U2U communication, where third-party providers are involved. The scope of this chapter is two-fold: on the one hand, it discusses the background and related work, specifically with regard to three third-party provider schemes: Centralized Service Provider (CSP), Federated Service Providers (FSPs) and providers that are peers in structured P2P overlay networks. On the other hand, it presents the scenario of U2U communication and presents an analysis of trust dependencies between the members of a closed group and the third-party provider(s). Also, the three schemes are compared with regard to criteria derived from the identified trust dependencies.

2.1 Model for User-to-User Communication

As introduced in Section 1.1.1, a U2U communication process equals the sharing of a *data object* among the members of a *closed group*. These two terms shall be discussed in more detail first. Afterwards it shall be discussed how the members use their device and how third partys are involved in sharing a data object. All aspects put together result in a novel, topic owner-based model to U2U communication which is used throughout this thesis.

2.1.1 Data Objects

A **data object** – as it will be understood throughout this thesis – is broadly defined as a generic container for human-readable information. This container can be interpreted by a specific **application**, running on a human user's **personal device**, in a way that the application can extract *data object's content* as the information and present it to the user for consumption. For example, a data object can be a self-contained file – ranging in size from a few bytes to multiple gigabytes – or be a part of an information stream, together with many other data objects. An example for the latter would be a chat session which consists of many single instant messages, with each instant message being a single data object to be interpreted by an instant messaging application and displayed to the user.

For the U2U communication scenario that will be regarded in this thesis, three central properties can be identified for each data object:

The first property is **confidentiality**: each data object conveys information that is not meant for public consumption, but only meant "for the eyes" of the members of an a-priori defined group of users, i.e. the *closed group*, which will be discussed below. Exposing a data object's content to anyone outside of the closed group is undesirable.

The second property is **linkability**: it is assumed that the identity of each data object's creator is inherently tied to the data object itself, thus giving the information inside the data object additional semantic. One example in Section 1.1.1 was a message with the content "Let's meet for dinner!", which is only useful to the reader when the original creator of the message is known to him. This is a contrast to the classic information retrieval scenario, where only the content of the data object is relevant but not its creator (e.g. a Wikipedia article).

The third property is **delay-tolerance**: it is assumed that a data object has no real time demands, in the sense that it is obsolete after a few milliseconds or seconds. One example for such a out-of-scope data object would be a voice-over-IP telephony packet. This is a necessary restriction for the scenario due to the way how User-Centric Networking reaches partition tolerance: here, consistency between devices is a secondary aspect and a delivery of data objects at a later time is acceptable. This issue will be discussed in Detail in Chapter 3.

2.1.2 Closed Groups

Each data object is associated to a **topic**. A topic is a specific semantic context, which is used for grouping data objects with the same audience. For example, a number of photos are different data objects. These photos can be grouped into a single photo album which is to be shared with the same users. In this case, the different data objects have the same topic.

Each topic has exactly one **topic owner**. The topic owner is one specific user that controls *access rights* for the topic: He selects for the given topic which other users are **allowed publishers** and which users are **allowed subscribers**. This selection is made from a universe of users – the topic owner's **contacts**. Two contacts are two users that have agreed to be potential partners for U2U communication. They are assumed to have mutually verified their identity earlier.

Allowed publishers are allowed to publish data objects, i.e. to become a publisher of a data object with the given topic. Only allowed subscribers have the option to indicate their interest in the given topic and become actual **subscribers**.

A closed group is associated to one specific data object δ_t with topic t. The members of the **closed group** assume one or multiple of the following roles:

- Topic owner of t
- Publisher of δ_t
- Subscribers of t

Figure 2.2 displays the relationships of these roles as a Venn diagram [88].

A closed group includes only those users that are actually meant to access the respective data object. Given that an allowed publisher publishes the data object δ_t with topic t, δ_t is meant to be distributed to all subscribers of t. Additionally, the topic owner of t may access δ_t. The group does not include any other allowed publishers or allowed subscribers. Another data object ϵ_t from another (allowed) publisher for topic t is associated to another closed group, consisting of the topic owner of t, the publisher of ϵ_t and the subscribers of t. Therefore, each closed group is defined on a **per-data-object basis**, not a per-topic basis.

Generally, there is no limit of how many users can be in a closed group. However, each closed group only consists of contacts of one single user (namely the topic owner) and mutual identity verification is an assumed prerequisite for being contacts, therefore a social relationship between the topic owner and each of his contacts is assumed. Examples here are friends, family, co-workers, etc. of the topic owner. Additionally, each closed group is associated to one specific data object – a user may share other data objects with his friends than he shares with his co-workers. Therefore, it is assumed that closed groups are very limited in size. A possible orientation value for an upper value is the median for Facebook friends of 99, as measured by [113].

Note the central role of the topic owner in a closed group, since all members are contacts of the topic owner. The publisher and a subscriber are not necessarily contacts. They may not even know each other. Also, the publisher is not necessarily aware which users are subscribers. A subscriber is not necessarily aware which other users are also subscribers. However, the identity of the publisher is exposed to the subscribers, due to each data object's property of linkability[1].

[1]A typical scenario in U2U communication where this situation occurs are comment sections. For example, a Facebook user A can comment on a post by user B. Both the post and the comment are visible to B and B's contacts (C, D,

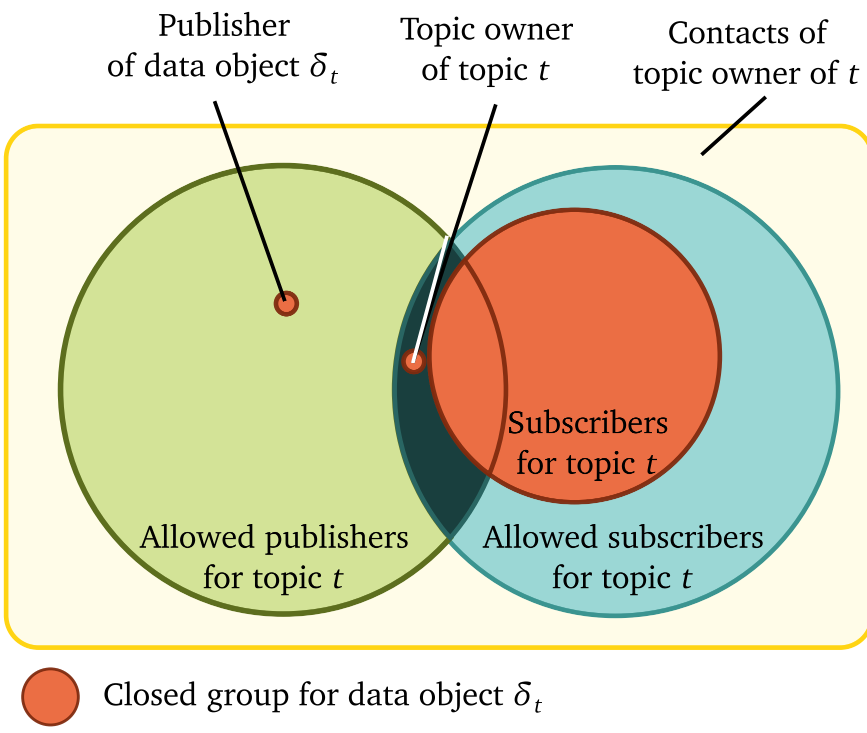

Publisher of data object δ_t
Topic owner of topic t
Contacts of topic owner of t
Subscribers for topic t
Allowed publishers for topic t
Allowed subscribers for topic t
Closed group for data object δ_t

Figure 2.2: *Venn diagram: closed group for a data object δ_t. The red subsets belong to the closed group, i.e topic owner of topic t, publisher of δ_t and subscribers of t*

2.1.3 User Identifiers

For a topic owner and a topic's subscribers, there needs to be a way to identify and select specific users:

- Topic owner: must be able to select a subset of users from his contacts to define allowed publishers and allowed subscribers for a topic he owns.
- Subscriber: link a received data object to its publisher

This can be solved with unique **user identifiers**. For example, two users that have agreed to be contacts can exchange their identifiers during the contact making procedure. The process for a topic owner to e.g. defining allowed publishers is then making a selection from the known user identifiers.

2.1.4 Personal Devices

Earlier, a data object was defined as a container that "can be interpreted by a specific *application*, running on a human user's personal *device*". These personal devices shall now be discussed in more detail.

. . .). However, if *B*'s contact list is not necessarily visible to *B*'s contacts, *A* does not know who else besides *B* can read his comment. Neither knows *C* that also *D* can read the comment, etc. Still, at least all these users can see that some user "*A*" has created the comment – whether they know *A* or not.

Each user has at least one device. Each device has exactly one user. Personal devices can be heterogeneous in terms of their resources (availability and capability, as will be discussed later in Section 3.2). They can range from smartwatches over smartphones up to PCs and small home servers (e.g. network-attached storages).

Different devices can have different applications running which are used for publishing data objects and consuming data objects, if the user is a subscriber.

With regard to later security and privacy considerations, each device is assumed to be fully trusted and controlled by its user. This means, that there are no backdoors, malware or any operating system features that might compromise the users' security or privacy. While this is an optimistic assumption for e.g. modern smartphone operating systems, such considerations are out of scope for this thesis and pose an orthogonal problem that remains to be solved.

Shared devices

For the sake of simplicity, it is assumed that each device is controlled by exactly one user. However, all considerations would be applicable for devices shared between users A, B, C, ... as well. Such a device would require the possibility to create different user profiles, letting one user log in at a time. While user A is logged in at that device, it is unavailable for the remaining users. This matches the situation where each remaining user would have a dedicated device which is currently switched off. As soon as user B logs in, A would "switch his device off". Such a multiplexing mechanism requires additional effort on the device's operation system layer which is not regarded here. Instead, multiple devices would be assumed.

2.1.5 Network Communication

In order to deliver a data object from a publishing device to a subscriber device, these devices need means to communicate with each other.

Two types of third parties can be involved here: **application providers** and **infrastructure providers**.

Third-party *application providers* are the relevant providers for this thesis. They operate on a "data object-aware" level, i.e. accept data objects from publishing devices and can store them intermittently or permanently and deliver them to subscriber devices. Additionally, they provide access control, i.e. let the topic owner define which user may be allowed publishers/allowed subscribers for a given topic.

Third-party *infrastructure providers* have the task to transport opaque bitstreams from one device to another, e.g. from publishing device to a server of the third-party application provider. They operate on a "data object-agnostic" level. Infrastructure providers can be local network administrators, consumer-level internet access providers up to tier 1 providers.

In the scope of this thesis – and specifically this chapter – are third-party application providers, not infrastructure providers. When **third-party providers** are mentioned, application providers are meant unless explicitly stated otherwise.

2.1.6 Summary

Figure 2.3 summarizes the scenario that was described in this section, by putting closed groups, data object, device, application provider and infrastructure providers together.

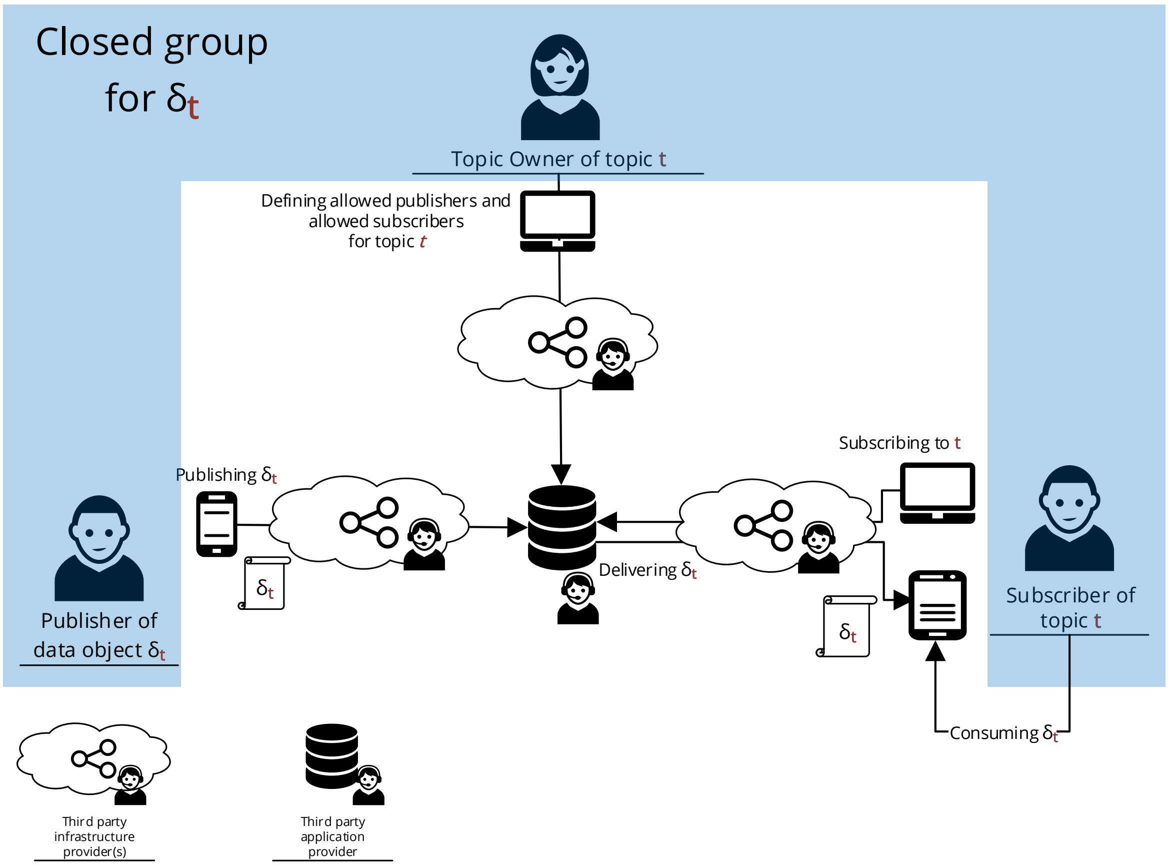

Closed group for δ_t
Topic Owner of topic t
Defining allowed publishers and allowed subscribers for topic *t*
Publishing δ_t
δ_t
Delivering δ_t
Subscribing to t
δ_t
Publisher of data object δ_t
Subscriber of topic t
Consuming δ_t
Third party infrastructure provider(s)
Third party application provider

Figure 2.3: *Overview: U2U communication scenario*

2.2 Third-Party Provider Schemes

A major problem in U2U communication deals with the question how data objects are delivered from the publishing device to the devices of the subscribers. In the state of the art, this problem is solved with the help of third-party providers. This section presents different *provider schemes* which involve one or multiple third-party providers. Each of them handles data delivery from the publishing device to the subscriber devices.

2.2.1 Data Object Delivery

Section 1.1.1 described the basic scenario in U2U communication: here, each *data object* gets created for a specific *topic* by a specific user – the *publisher*. The data object shall now be delivered to a specific set of target users – the *subscribers*.

The creation of the data object is carried out by an application which runs on one device of the publisher – the **publishing device**. It finally gets delivered to one or multiple devices of each subscriber, i.e. the **subscriber devices**.

Although publishing device and subscriber device could communicate directly, involving third-party providers constitutes the state of the art in U2U communication. For this process – getting a data object from the publishing device to a subscriber device, including at least one third-party provider – the following workflow is regarded:

(1) The publishing device *forwards* the data object to a specific provider device a of a third-party provider A.

(2) *(Only for decentralized provider schemes; optional)* If a decentralized provider scheme is used, multiple third-party providers can be involved. In this case, the provider device a *forwards* the data object to one or multiple provider device(s) $b, c, \ldots$ of other third-party providers $B, C, \ldots$ This step might be repeated by $B, C, \ldots$ forwarding the data object to other providers. How this step is executed, depends on the decentralized provider scheme.

(3) A provider device which holds the data object (i.e. by forwarding from the publishing device or another provider device) subscriber device *delivers* the data object.

Note that publishing device and subscriber device do not communicate directly with each other. Furthermore, a copy of the data object is stored on the provider device outside of the control of any personal device.

Figure 2.4 depicts the process of data object delivery.

2.2.2 User Identifiers

In order to perform the described tasks, third-party providers need an understanding of the respective access rights, as defined by the topic owner. Since the topic owner operates on limited pool of users known to him, i.e. his contacts, he makes his selection of allowed publishers and allowed subscribers via user identifiers.

Given that the topic owner passes user identifiers directly to the third-party provider (possible alternatives will be discussed in Section 2.3.2), the following checks have to take place by the third-party provider:

- **Allowed publishers**: When a publishing device forwards the data object to a third-party provider, the corresponding publisher must have the rights to do so as given by the topic owner. The provider scheme must provide a mechanism for the topic owner to *define allowed publishers*. Additionally, the provider scheme must provide a mechanism where at least one provider is responsible for checking the rights of the publisher before data objects get delivered to the subscribers.

- **Allowed subscribers**: In order for a user to successfully subscribe to a specific topic, he must have the rights to do so as given by the topic owner. The provider scheme must provide a mechanism for the topic owner to *define allowed publishers*. Additionally, the provider

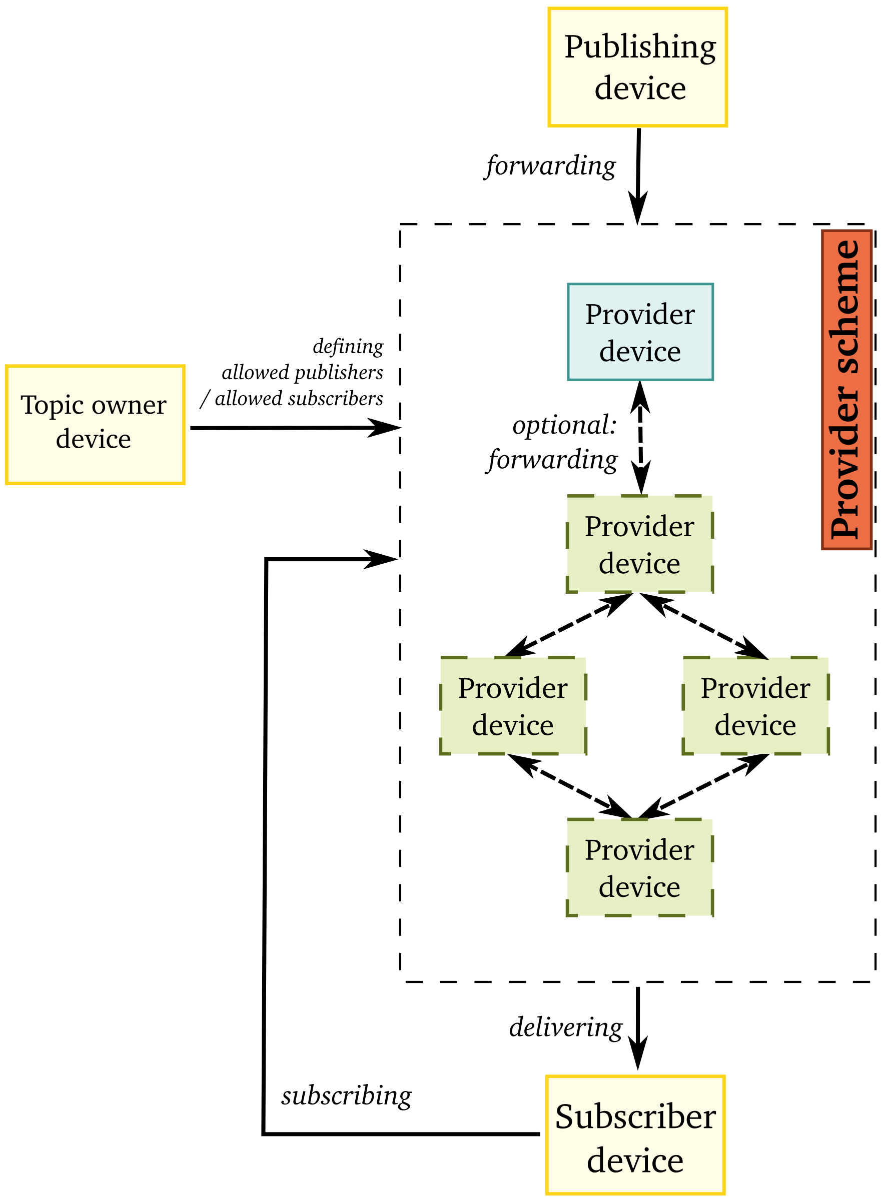


Figure 2.4: *Overview: user devices and provider scheme*

scheme must provide a mechanism where at least one provider is responsible for checking the rights of the user that placed the subscription request.

- **Subscriptions**: In order to deliver a data object to the correct target users (i.e. the subscribers) and only these, the third-party provider needs to know the mapping between topic and subscribers. The provider scheme must provide a mechanism for the users to *subscribe*. The provider scheme must ensure that data objects get delivered to subscribers only.

Thus, third-party providers have two central tasks: *access control* and delivering data objects to the correct subscribers. They need to be able to distinguish users from each other, e.g. to identify a subscriber from a non-subscriber. Typically, this problem is also done by unique **user identifiers** – similar to the situation inside the closed group (see Section 2.1.3).

Examples for user identifiers in U2U communication with third-party providers are as follows:

- An SMTP server accepts an email and forwards it to the target SMTP server based on the target (recipient) email address. Here the email address is an identifier which lets the email reach the correct destination.
- A user publishes a private photo album with restricted access by using a popular centralized *online social network (OSN)*. Each user in the OSN has his own profile and user ID. Here the user IDs are identifiers to define access to the photo album.
- An instant messaging service for mobile phones identifies its users by their mobile phone number. A user can upload his full address book to the service to determine which of his phone book entries are also users of the service. Here, the phone numbers are identifiers which not only let an instant message reach the correct destination (similar to the email example), but also let the user discover new contacts on the service.

2.2.3 Advantages of Third-Party Providers

The reason for using a third-party provider in the first place is data availability: first, the published data object from the publisher's personal device is forwarded to and stored at the third party. Second, the data object gets delivered from the third party to the subscriber's personal device. This way, the two personal devices do not have to communicate directly with each other. This eliminates the requirement that the two personal devices are available for each other at the same time. Besides delivering the data object, the third-party provider can also permanently persist the data object in case it gets deleted from both personal devices, but is still needed at a later time.

In the following, three common third-party provider schemes for U2U communication are presented. Specifically the role of user identifiers will be discussed with the help of an use-case example, since these are relevant for later considerations about application metadata privacy.

- Centralized Service Provider (Section 2.2.4)
- Federated Service Providers (Section 2.2.5)
- Structured P2P Overlay Networks (Section 2.2.6)

2.2.4 Centralized Service Providers

Nowadays, Centralized Service Providers (CSP) are the most common provider scheme for U2U communication: in September 2015, *Facebook* had on average 1.01 billion daily active users [34]. *Snapchat* [104] is an ephemeral instant messaging app which lets publishers set a time limit for how long subscribers can view published photos and videos (Snaps). Snapchat claims that 6 billion Snaps were published in November 2015, three times the number of Snaps from May 2015 [37].

As the name indicates, a CSP is both *centralized* and a *service provider*:

- Centralized: the CSP stores all relevant data of all users of a given application. For this reason, the CSP is seen as centralized here. This does not necessarily apply to the network topology: depending on the CSP's size, the server platform does not consist of a single machine, but of large data centers that may even be distributed across different countries. Therefore, a distinction has to be made between **authorial centralization** and **technical centralization**. Authorial centralization means "*that one single entity (e.g. company) has the omnipotent power to decide about what is allowed on their [. . .] platform*" [79]. For the considerations in this thesis, only authorial centralization is relevant.

- Service Provider: A CSP does not provide an application-agnostic platform to its customers, but a specific U2U communication application or a specific set of applications. Examples for a CSP are as mentioned aboved Facebook [33] and Google Plus [43] (online social networking, photo storage and sharing), WhatsApp and Snapchat [116] (messaging), Google Docs [44] (collaborative work) or Dropbox [29] (online data storage and file-sharing).

A CSP is often a company with a high interest in profit and providing a satisfying customer experience. To achieve this, CSPs need to invest in data centers with powerful, redundant hardware.

Usually, a CSP does not bind a user to a device, but instead allows its users to use any device to make use of the service. Before using the service however, the user has to login at the service with a unique user identity and private credentials.

Case study: Sharing a photo on Facebook

Figure 2.5 shows a communication process, where Alice publishes a data object (photo) on Facebook with Bob and Charlie being subscribers. Earlier, she created an photo album "Alice's photos" and gave Bob and Charlie – contacts of Alice – read access via their user identifiers in the respective Facebook privacy settings. In the model for U2U communication (Section 2.1), this album has an according topic and Alice is the topic owner. Bob and Charlie are allowed subscribers.

For publishing the photo, she uses her device a_1, whereas she did the album creation with device a_2. However, Facebook recognizes her as the same user due to a mandatory login with private user credentials tied to a unique user identity.

By default, allowed subscribers are automatically subscribers on Facebook, unless they have proactively chosen to hide or block specific content. Hence, Bob and Charlie are subscribers and when they browse their Facebook the next time with one of their devices, Alice's photos is delivered to the respective devices.

2.2.5 Federated Service Providers

Like CSPs, FSPs are service providers, i.e. they provide a specific U2U communication application or a specific set of applications. In contrast to CSPs, Federated Service Providers (FSPs) are authorially decentralized: instead of only one provider for a given application, there are multiple,

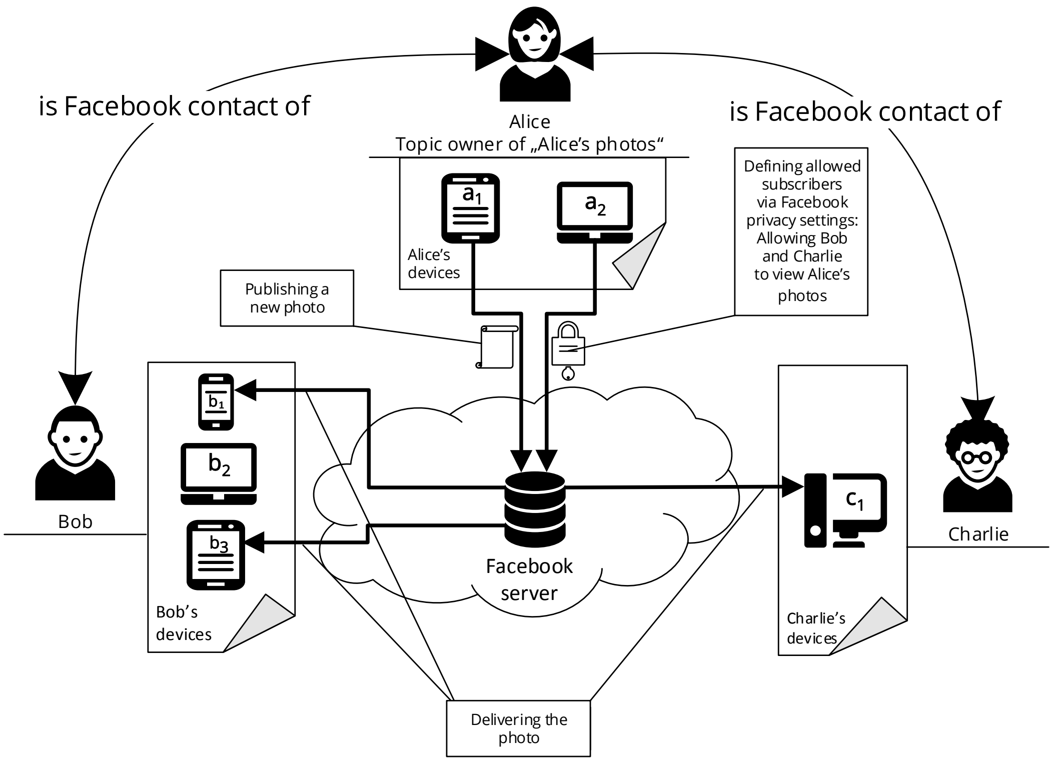


Figure 2.5: *CSP case study: Sharing a photo on Facebook*

independently controlled service providers. Each user can choose his own service provider and can migrate to another one without affecting his access to the given application nor other users.

All FSPs for the same application agree on a *federation protocol* [79] for inter-provider communication. New FSPs can emerge and seamlessly communicate with already existing FSPs.

A suitable addressing scheme is required which identifies the user and his service provider. When a data object is published, it can be forwarded to the correct provider first, who then – analog to a CSP – can identify the correct subscriber and deliver it to the subscriber's personal device.

Popular application examples that use FSPs are e-mail [64], XMPP (messaging) [90] [91] and *Diaspora* (social network profiles) [11].

Case study: instant messaging via XMPP/Jabber

Figure 2.6 shows Bob using his personal device b_1 to send an instant message to Alice. He uses XMPP [90] to do so, which is an FSP-based protocol/application.

Bob is a customer of the service provider "beta.com" and his address "*bob@beta.com*", which contains Bob's provider-wide unique user identity and the provider. The same applies to Alice with "*alice@alpha.org*".

First, Bob authenticates at "beta.com" with his unique identity and private credentials. The server (provider device) of "beta.com" ensures therefore that not any user can publish a message in Bob's name. Bob's device can now forward the message. Bob needs to address Alice and her

FSP via "*alice@alpha.org*" which can be interpreted as the message's topic. This server identifies Alice as a subscriber (since she is the addressed target of the message). Optionally, the server can again identify Bob as an allowed publisher, e.g. if Alice maintains a whitelist of allowed senders on "alpha.org". Finally, the server of "alpha.org" delivers the instant message to Alice's devices.

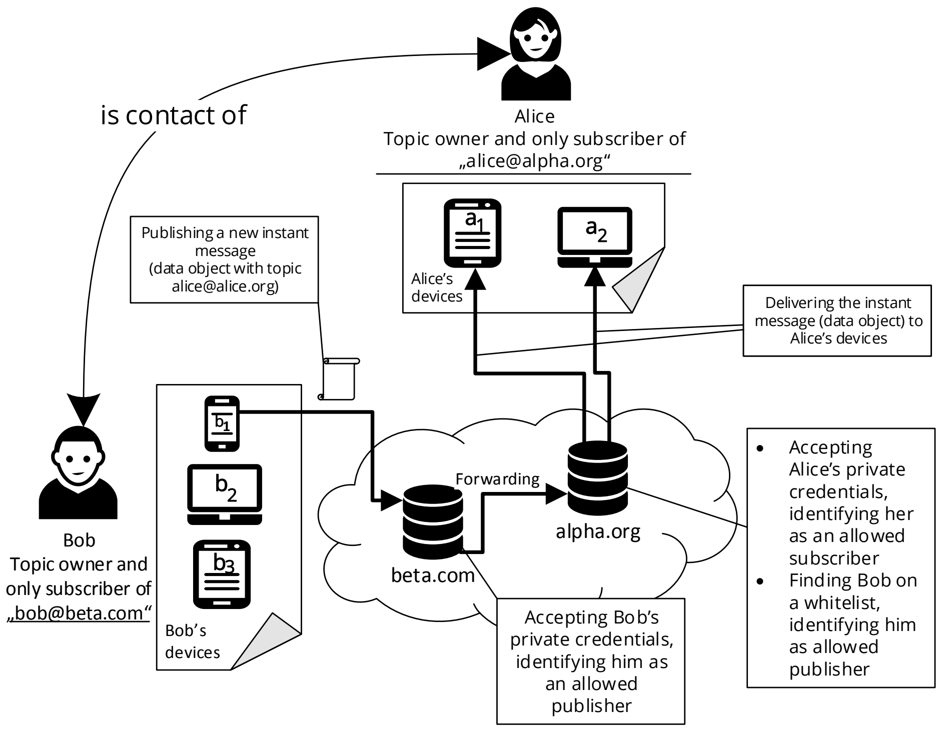

is contact of
Alice
Topic owner and only subscriber of „alice@alpha.org"
a1
a2
Alice's devices
Publishing a new instant message (data object with topic alice@alice.org)
Delivering the instant message (data object) to Alice's devices
b1
b2
b3
Bob's devices
Bob
Topic owner and only subscriber of „bob@beta.com"
Forwarding
beta.com
alpha.org
Accepting Alice's private credentials, identifying her as an allowed subscriber
Finding Bob on a whitelist, identifying him as allowed publisher
Accepting Bob's private credentials, identifying him as an allowed publisher

Figure 2.6: *FSP case study: Instant messaging via XMPP/Jabber*

2.2.6 Structured P2P Overlay Networks

Structured P2P overlay networks are a well-researched type of decentralized network storage and communication [39] [54] [71] [82] [84] [87] [106] [119] . Contrary to CSP/FSPs, the data objects are not stored on one or a few highly-available servers, but are evenly distributed on all participating personal devices (*peers*).

[69] defines a structured P2P overlay network as follows:

> "*[A structured P2P] overlay network assigns keys to data items and organizes its peers into a graph that maps each data key to a peer. This structured graph enables efficient discovery of data items using the given keys.*"
>
> [...]
>
> "*Structured P2P systems use the Distributed Hash Table (DHT) as a substrate, in which data object (or value) location information is placed deterministically, at the peers with identifiers corresponding to the data object's unique key. DHT-based systems*

> *have a property that consistently assigned uniform random NodeIDs to the set of peers into a large space of identifiers. Data objects are assigned unique identifiers called keys, chosen from the same identifier space. Keys are mapped by the overlay network protocol to a unique live peer in the overlay network. Given a key, a store operation (put(key,value)) [or a] lookup retrieval operation (value=get(key)) can be invoked to store and retrieve the data object corresponding to the key, which involves routing requests to the peer corresponding to the key.*
>
> *Each peer maintains a small routing table consisting of its neighboring peers' NodeIDs and IP addresses. Lookup queries or message routing are forwarded across overlay paths to peers in a progressive manner, with the NodeIDs that are closer to the key in the identifier space.*"

Therefore, each peer is responsible for storing a fraction of the data objects in the system. All stored data objects get evenly distributed over the participating peers. Consequently, if a structured P2P overlay network is used as the provider scheme in U2U communication, personal devices have to act as provider devices. This is a strong difference to the previously described CSP and FSP approaches where provider devices are dedicated, highly available servers. Since responsibilities are defined by closeness between a key and a random NodeID, a personal device does not necessarily store data objects where the user is publisher or subscriber.

In contrast to highly available servers, personal devices might become unavailable to other peers. If a data object is only stored on one peer, it cannot be retrieved if the storing peer is unavailable. As a consequence, data objects do not only get stored on one peer, but are replicated to the k peers with the k next closest NodeIDs. If one of these peers becomes unavailable for the other peers, the data object gets replicated to the next available node chosen by a protocol-specific metric. Similarly, if a node with a NodeID becomes available (again) with a closer NodeID than the replication node with least closest NodeID, the new node replaces the old one in the set of the k nodes. This way, it is ensured that each data object is replicated on k nodes as long as at least k nodes are available for other peers.

This replication comes at a cost: The load on the overlay network and therefore single peers increases with higher k and larger data objects. When a responsible peer becomes unavailable, available, unavailable again, etc. (i.e. it suffers from *churn*), replicas may be shifted back and forth between the k closest peers. The higher the churn is, the more often this maintenance can happen.

Unlike a CSP or FSP which provides a dedicated U2U communication service, a DHT as a distributed data storage does not have a notion of a user identifier. However, if it is used for storing data of U2U communication applications, the device's NodeIDs are inherently tied to processes like store and lookup operations. A node which is responsible for storing a specific data object can track which NodeIDs access said data object. Additionally, in systems like S/Kademlia [9], the NodeID is the hash of a public key, which is an approach to prevent *eclipse attacks* [102], i.e. the free choice of a NodeID to enforce a specific placement in the overlay network or claim a specific responsibility. Depending on the key management, a public key can also be used for user identification.

General workflow

Figure 2.7 shows a U2U communication example with a structured P2P overlay network as the provider scheme. The given overlay network consists of 12 devices. Alice's, Bob's and Charlie's personal devices participate in the network and hence belong to the 12 devices.

Alice decides to publish a photo (data object δ). She generates a key k_δ for the value δ. When forwarding δ under the associated key k_δ, the DHT protocol determines node r as responsible for k_δ and hence storing δ. δ gets routed via multiple hops (the dotted arrows in Figure 2.7) to r.

Given that Bob and Charlie know k_δ, they can use their devices b_1 and c_1 to *get(k_δ)* and retrieve it from r.

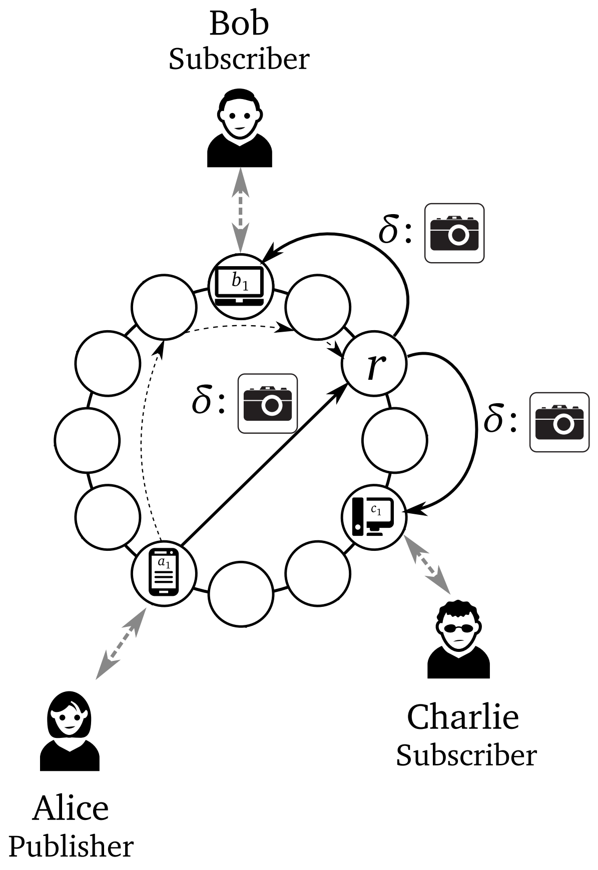


Figure 2.7: *Basic Distributed Hash Table workflow*

Case study: BitTorrent Sync

BitTorrent Sync is similar to cloud-based data storage / file-sharing application similar such as Dropbox or Google Drive. Such CSP-based applications "*allow users to store their data in a virtual extension of their local machine with no direct user interaction required after installation. It is* *also backed up by a full distributed data-centre architecture*[2] *that would be completely outside the financial reach of the average consumer. Their data is available anywhere with Internet access [. . .] Some services such as Dropbox, also have offline client applications that allow for synchronisation of data to a local folder for offline access.*" [36]

BitTorrent Sync implements this type of application in a decentralized way which also involves a DHT. Instead of storing a data object at a CSP, it is replicated and kept synchronized on a set of personal devices which are subscribed to the same *ShareID*. BitTorrent Sync is a closed-source protocol and has been researched in [36].

Each ShareID has to be explicitely subscribed by each device that shall receive the associated data objects, i.e. files or folders. Between different users, the ShareID must be communicated a-priori via an additional channel such as email or instant messaging. All involved devices need to find each other for delivering data objects, i.e. know each other's IP address and port. BitTorrent Sync provides several options to solve this problem (see [36] for details), one of them is a structured overlay P2P network/DHT. Each peer stores its peer details *<SHA1(ShareID):IP:Port>*as a data object on its own in the DHT. These peer details can be retrieved by other peers that know the ShareID. On the other hand, the responsible node for a ShareID can track which NodeIDs access the information about a ShareID.

For performance reasons, the file replication itself is performed with the BitTorrent protocol [19] directly from one device of the synchronization group to another. BitTorrent is designed with large files in mind, putting these into the DHT would result in high maintenance effort for replications. Peer details on the other hand are small in size and can be replicated with low effort.

2.3 Security and Privacy in U2U Communication

In the following, four security and privacy goals are explained that are relevant for U2U communication. While there are other goals, these four are deemed the most central goals for the scope of this thesis and are therefore discussed in detail.

- *Content data confidentiality* (Section 2.3.1)
- *Content data integrity* (Section 2.3.1)
- *Content data availability* (Section 2.3.1)
- *Application metadata privacy* (Section 2.3.2)

With regard to these goals, trust dependencies can be identified. Trust is discussed in Section 2.3.3.

2.3.1 Content Data Security

Content data is the payload of a data object, i.e. the actual information to be given from a publisher to the subscribers. The information conveyed in content data is the reason why the data object was created in the first place.

[2]Note that the authors refer to *technical decentralization* here, not *authorial decentralization* – see Section 2.2.4 about this difference in CSPs.

In Section 2.1.1, it was defined that one property of a data object is confidentiality: a data object is meant to be shared between the members of a closed group only. The content data conveyed within the data object is not meant for the eyes of third parties. Therefore, **content data confidentiality** is a security key concept which plays an important role here.

Additionally, content data is supposed to be eventually accessed by all subscribers. Access for one specific, multiple or all subscribers must not be maliciously prevented. This requirement is especially relevant for the provider devices, as these hold data objects meant to be delivered to subscriber devices. Therefore, **content data availability** is another important security key concept.

Lastly, a data object is meant to be accessed by all subscribers in the way the publisher created and published it. It must not be maliciously modified by any group member or third party: **content data integrity** is the third security key concept which is has to be taken into account. At least *weak integrity* is desirable here, where manipulation can be detected in hindsight. *Strong integrity* (preventing manipulation in the first place) is not regarded here.

2.3.2 Application Metadata Privacy

In the context of this thesis, **metadata** is data that is not content data, but is required within the context of any communication process (*explicit*) or is an incidental by-product thereof (*implicit*). **Explicit metadata** *is required to establish a communication process in the first place.* Most notable examples for explicit metadata are all types of addresses: cellular network caller IDs, MAC addresses, IP addresses, NodeIDs, email addresses, URLs, etc.

With regard to U2U communication, *user identifiers* are an important type of explicit metadata, as discussed in Section 2.1.3. For example, user identifiers are used by the topic owner to define allowed subscribers for a topic he owns. This definition must be done before data objects with regard to this topic can be published.

There are now two possibilities: either the topic owner directly passes the respective target user identifiers to the providers or some kind of attribute-based access control is used. In the latter case, the relevant attributes also need to be applied by the topic owner to the target users first, before the U2U communication can take place. Thus, in both cases a communication process needs to be performed, which links the topic owner to the target users.

Another alternative would be to perform no access control at all: each data object is accessible to everyone, but is encrypted in a way that only subscribers can read it. In that case, a key exchange between topic owner and subscriber needs to be performed first. Also, the topic owner needs an out-of-band mechanism to announce new data objects to the subscribers. Given that they also rely on providers here, at least an initial linkage between the users (e.g. by user identifiers) is required to enable the U2U communication and is therefore explicit metadata.

By repeated service usage, **implicit metadata** incidentally accumulates at a service provider. Examples for implicit metadata with regard to the cases above are:

- Contact frequency and inter-contact times between two email users A and B, measured by the number and timestamps when A sends an email to B or vice-versa

- Number and frequency of accesses on the photo album by different contacts, indicating the popularity of the publisher and his photos.
- Reconstructing social networks by matching address book entries of different users.

Explicit and implicit metadata needs to be taken into account for privacy considerations if it lets an attacker draw conclusions about the identity, preferences, social connections or behavioural patterns of a given user.

The most critical type of metadata gives away **personal identifiable information (PII)**, which is "*information which can be used to distinguish or trace an individual's identity, such as their name, social security number, biometric records, etc. alone, or when combined with other personal or identifying information which is linked or linkable to a specific individual, such as date and place of birth, mother's maiden name, etc.*" [63]. User identifiers can easily leak PII: for example, email addresses are often not random aliases, but are deliberately tied to a user's identity in order to be memorable (e.g. "fabian.hartmann@kit.edu"). An OSN's terms of service often demand from the users to provide their real name. A mobile phone number is closely linked to a single natural person.

Even if a user's identity is not given away directly by PII, both explicit and implicit metadata in U2U communication can lead to assumptions about the users and thus be a privacy threat. Random user identifiers, inter-/intra-contact times, upload and download timestamps, the frequency of data object exchanges inside the closed group, the size of data object δ_t etc. can be used to draw conclusions about the closed group even if the content data of δ_t is unknown. Such conclusions can be assumptions about the social closeness of two users, the interest in one specific data object, etc.

Section 2.5.1 presents related work that discusses the impact of metadata on privacy in more detail.

Metadata on Different ISO/OSI Layers

So far, the focus was put on U2U communication and third-party providers for application services. Mapped to the ISO/OSI layer model, the previous considerations are leveled on layer 7 – the application layer. Thus, the regarded types of metadata are called **application metadata**. However, similar considerations about explicit and implicit metadata can be made for the network layer.

This metadata on the network layer (relevant for infrastructure providers) is called **communication metadata**. Communication metadata is relevant on the routes between the communicating parties through intermediate networks, i.e. the routes between publisher and third-party provider and between third-party provider and subscriber.

Assume two devices a and b which communicate using the Internet Protocol (IP). An IP router forwards an IP packet based on the destination IP address in the packet's header. Additionally, the packet header contains the source IP address. By analyzing these addresses, an IP router which forwards packets between a and b can analyze communication frequency, traffic size and inter-contact times between a and b, measured by the number and timestamps when an IP packet from a to b passes this router.

Similarly, a telephone company infers which person calls which number for how long, can log these information and e.g. provide it to governmental authorities.

Therefore, similar privacy considerations apply to connection metadata as to application metadata. However, connection metadata is not in the focus of this thesis.

2.3.3 Trust

A central aspect in every privacy-aware system is **trust** between the involved entities. In the following, a definition for trust is given. Afterwards, the trust dependencies between a closed group and a third-party provider, as well as between the group members themselves are discussed.

Trust definition

[100] gives a standard definition for trust in computer science as "*a subjective expectation an entity has about another's future behavior*".

Based on this generic definition, the following definition is derived for the U2U communication scenario: *Closed group member X's trust in another party Y (Y being another closed group member or third-party provider) is X's subjective expectation that Y does not take advantage of data affecting X beyond Y's task in the communication process – in terms of content data confidentiality, content data availability, content data integrity and application metadata privacy.*

[22] differentiates between three types of trust:

> - *Blind trust: [...] The strongest form of trust; from a technical point of view it could lead to the weakest solutions, the ones most vulnerable to misplaced trust*
> - *Verifiable trust: [...] Trust is granted by default but verifications can be carried out a posteriori (for example using commitments and spot checks) to check that the trusted party has not cheated.*
> - *Verified trust: [...] a "no trust" option; it relies on cryptographic algorithms and protocols (such as zero knowledge proofs, secure multiparty computation or homomorphic encryption) to guarantee, by construction, the desired property. Furthermore the amount of trust necessary can be reduced by the use of cryptographic techniques or distribution of data.*

It will now be discussed for the four security and privacy goals whether blind trust is required or a weaker form suffices. This discussion will be made with regard to third-party providers and group members respectively. If party A trusts in party B, it has the following meanings:

- **Content data confidentiality:** A trusts B that B does not pass the content of a confidential data object to a party outside the closed group, i.e. a party which is not meant to access said data object.
- **Content data integrity:** A trusts B that B delivers a specific data object to A correctly, i.e. identical to when it was originally published.

- **Content data availability:** A trusts B that B delivers a specific data object to A.
- **Application metadata privacy:** A trusts B that B does not disclose information about A to a party which is not meant to learn about A's involvement in a specific U2U communication process.

Trust dependencies

For the following, assume a closed group $\mathbb{G}_{\delta_t}$ for data object δ_t. The members of $\mathbb{G}_{\delta_t}$ are: the **topic owner (TO)** T of topic t, the publisher P of δ_t, and one subscriber S of topic t. For a clearer separation of roles, assume here that T is not a subscriber of t. A third-party provider scheme is used, i.e. at least one third-party provider Z is involved.

In the following, it is first discussed whether the members of $\mathbb{G}_{\delta_t}$ have to trust Z in the four security and privacy goals. Afterwards, it is discussed whether the members of $\mathbb{G}_{\delta_t}$ have to trust each other in this regard.

Trust in third-party providers

In the following, it is discussed for the four security and privacy goals if the members of $\mathbb{G}_{\delta_t}$ need to trust Z.

- **Content data confidentiality**: Z controls a provider device which accepts δ_t from P and delivers it to S. Given that sufficiently strong **end-to-end encryption** is used, P can encrypt δ_t in a way that only S can read it.[3] Z or any other party cannot decipher δ_t and confidentiality is ensured. In this case, $\mathbb{G}_{\delta_t}$ does not have to trust Z with regard to confidentiality.

  If P did not encrypt δ_t in a way that only S can read it, δ_t is readable by Z. In this case, $\mathbb{G}_{\delta_t}$ needs to blindly trust Z that Z does not exploit the information conveyed in δ_t.
- **Content data integrity**: $\mathbb{G}_{\delta_t}$ relies on Z to deliver δ_t from P to S with exactly the same content P created. Z must not modify δ. While encryption can be used to ensure content data confidentiality, *cryptographical signatures* or *Message Authentication Codes (MAC)* can be used to detect modifications and therefore ensure weak content data integrity. When e.g. using signatures, P cryptographically signs δ_t, making modifications to δ detectable to S as long as S has the possibility to check the signature. If no such mechanism is used, $\mathbb{G}_{\delta_t}$ must blindly trust Z to maintain integrity.
- **Content data availability**: The closed group $\mathbb{G}_{\delta_t}$ relies on Z to deliver δ_t from P to S. Instead of delivering δ_t to S, Z might delete δ_t or deny access to it. Reasons for such a behaviour on the provider's end can be of accidental nature (e.g. due to data loss) or intentional (e.g. censorship). Therefore, the members of $\mathbb{G}_{\delta_t}$ have to trust Z in regard to content data availability. Depending on the application, this trust can be verifiable – if missing data objects are detectable by the application or attract the users' attention. As

[3] A good introduction to cryptography is for example [93]

soon as Z appears unreliable in terms of availability, $\mathbb{G}_{\delta_t}$ would probably try to change the service or provider.

- **Application metadata privacy**: As discussed in Section 2.3.2, a third-party provider is given a mapping between topic and users in order to enforce access control. Therefore, application metadata is made visible by $\mathbb{G}_{\delta_t}$ to Z in order to use the service. Explicit and implicit application metadata lets Z identify the members of $\mathbb{G}_{\delta_t}$ or at least learn about their communication patterns. Z can use and further process this information without $\mathbb{G}_{\delta_t}$ noticing it. Therefore, $\mathbb{G}_{\delta_t}$ needs to blindly trust Z not to exploit their application metadata.

Trust in group members

In the following, it is discussed for the four security and privacy goals if the members of $\mathbb{G}_{\delta_t}$ need to trust each other.

- **Content data confidentiality**: Since δ_t is confidential, each member of $\mathbb{G}_{\delta_t}$ has to blindly trust other members to not disclose δ_t to non-members:
  - P and S have to trust each other. Both P and S have access to δ_t. P created δ_t in the first place and S is a subscriber of t and therefore supposed to read δ_t. Considerations about DRM are made further below.
  - Both P and S have to trust T. Topic owner T can define himself as a subscriber anytime, therefore access to δ_t is possible for T. Note that P might not know which other users are subscribers and thus receive δ_t. P's trust in S is therefore transitive from P's trust in T – P assumes that T will make a good selection of allowed subscribers.
  - T has to blindly trust both P and S. T has to blindly trust P and S: even if T does not subscribe to t, but only defines allowed publishers and allowed subscribers, T has probably an interest that confidential data objects that get published under his topic t stay confidential.

  Keeping content data confidential inside the closed group is more of a social than a technological problem. If the members of $\mathbb{G}_{\delta_t}$ cannot blindly trust each other in this regard, the idea of sharing a confidential data object is flawed to begin with. *Digital Rights Management (DRM)* is a technical attempt to control this issue by "*[managing] the usage of content once it has been traded*" [112]. This means that "*DRM enables, for example, to restrict the consumption of digital content to a predefined timespace, to limit the number of times digital content can be consumed, or to specify the allowed actions (e.g. view, but not print)*" [74]. However, these are technical limitations regarding the data object, not the conveyed information between humans. Once information has been disclosed to a user, it is by intuition very hard to prevent him from (ab)using this knowledge – at least by technical means.

- **Content data integrity**: The members of $\mathbb{G}_{\delta_t}$ do not have to trust each other when a third-party provider is used. In the given case, only Z is responsible for storing and delivering δ correctly to the subscribers.

- **Content data availability**: The members of $\mathbb{G}_{\delta_t}$ do not have to trust each other when a third-party provider is used. In the given case, only Z is responsible for storing and delivering δ to the subscribers.

- **Application metadata privacy**: Metadata privacy becomes interesting if not only the content data of δ_t may be kept confidential inside the closed group $\mathbb{G}_{\delta_t}$, but also the communication process itself. If one group member X can identify another group member Y as a member of $\mathbb{G}_{\delta_t}$, Y has to blindly trust X that X does not disclose Y's memebership. Knowledge about other members is as follows:

  - T **knows about** (P, S): T knows that P and S are potential members of the group. As the topic owner, T allowed P (S) to be an allowed publisher (subscriber) in the first place. Allowed subscribers are not necessarily actual subscribers and it depends on Z to provide actual subscriber information to T.

  - (P, S) **know about** T: Both P and S know that T is the topic owner and therefore T is member of the group. P knows that he publishes a data object to a topic owned by T. S knows that he subscribed to a topic owned by T.

  - S **knows about** P: In Section 2.1, it was assumed that each data object is linkable to the publisher. The publisher is identifyable for each subscriber. Therefore, P is known to S.

  Due to the pub/sub model, the following two restrictions apply:

  - P **does not know about** S: Note that the set of allowed subscribers is defined by T, not P. P publishes δ_t for a topic t where P knows that T owns t. On publishing, P passes δ_t to Z. Z is responsible to deliver δ_t to S, but P is not necessarily aware which user is S or if there are any other subscribers. This restriction does not apply in cases where publisher and topic owner are the same user. Additionally, for applications where it is important for P to learn about all subscribers (e.g. a group chat), this information can be shared by the application itself.

  - S_1 **does not know about** S_2: Z is responsible to deliver δ_t to S_1 and S_2, but S_1 is not necessarily aware that S_2 is another subscriber and vice-versa. For applications where it is important for subscribers to learn about all other subscribers (e.g. a group chat), this information can be shared by the application itself.

  Thus, due to the linkability of the data object to its publisher, subscribers can identify the publisher by his user identifier. Thus, there is **no sender anonymity** in the given U2U communication scenario.

  Since the allowed subscribers are defined by the T only, neither P nor S_1 necessarily knows that S_2 is a subscriber. Thus, **receiver anonymity** for S_2 towards P and S_1 is generally possible unless T decides to share that information.

Summary

This section analyzed trust dependencies in regard to four the different security and privacy goals (Section 2.3.1 and 2.3.2):

- **Content data confidentiality:** Each group member has access to the shared data object, therefore all group members have to blindly trust each other to keep it confidential. When using end-to-end encryption, the group members do not have to trust the third-party provider here. Otherwise, blind trust by the group members in the third-party provider is required.
- **Content data integrity:** If signatures or MACs are used, the group members do not have to trust the third-party provider here. Otherwise, blind trust by the group members in the third-party provider is required. Since the group members themselves are not responsible for delivering the data object, they do not have to trust each other here.
- **Content data availability:** The group members need to trust the third-party provider to deliver the data object. Since the group members themselves are not responsible for delivering the data object, they do not have to trust each other here.
- **Application metadata privacy:** The group members need to blindly trust the third-party provider here. The group members also need to trust each other, with one exception: the subscriber does not have to trust the publisher, since the publisher does not know which users are subscribers. On the other hand, the publisher is known to the subscribers. Thus, the publisher needs to *transitively trust* all subscribers – the subscribers are unknown to the publisher and yet he has to trust them, therefore the publisher trusts that the topic owner has made a "good selection" of allowed subscribers.

Figure 2.8 summarizes these considerations.

2.4 Comparison of Third-Party Provider Schemes

In this section, the three third-party provider schemes presented in Section 2.2 are compared. For comparison, four different criteria are regarded. These criteria are based on the trust dependencies between closed group members and third-party providers which were identified as critical in Section 2.3.3: application metadata privacy and content data availability. In Chapter 3, User-Centric Networking will be discussed. User-Centric Networking aims at removing third-party providers from U2U communication. For the sake of comparability with the three third-party provider schemes, it will be discussed with regard to these criteria as well.

2.4.1 Criteria for Comparison

The following five criteria are regarded: application metadata privacy, vendor lock-in, connectivity constraints, required infrastructure and resource strain on personal devices. The first three criteria are based on the trust dependencies privacy and availability which were identified as critical in Section 2.3.3, since the group members need to trust third-party providers here. The latter two

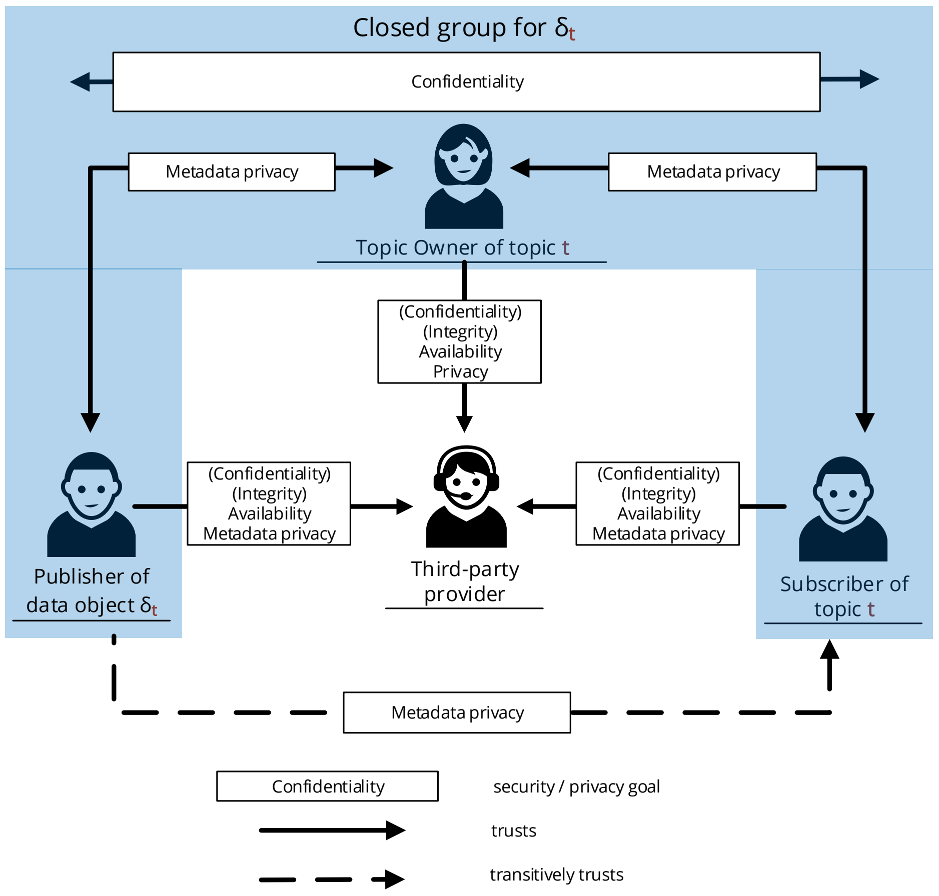


Figure 2.8: *Trust dependencies between group members and third-party provider*

criteria regard the required resources in order to establish a provider scheme. All five criteria will be revisited for User-Centric Networking in Chapter 3 and allow User-Centric Networking to be compared to the three third-party provider schemes.

Application Metadata Privacy

This criterion deals with the different privacy implications for each provider scheme. In Section 2.3.2, it was established that application metadata of closed group members gets exposed to third-party providers. Therefore, Section 2.3.3 presented a blind trust dependency with regard to application metadata privacy from closed group members to third-party providers. This applies to all three provider schemes, which is a negative aspect of all three and the inspiration of User-Centric Networking. Nevertheless, the different privacy threats shall be discussed for each provider scheme.

Vendor Lock-In

This criterion deals with the question how strongly a provider scheme is tied to an application. How easy it is for a user to change the provider without giving up the application altogether? What happens to the application if a provider shuts down permanently? Assuming that providers may come and go, a group of users can ideally use a specific U2U application as long as they want without being dependent on the existence of a specific provider. Changing to another application results in overhead for the users such as agreeing on an alternative application (if any exists) or re-importing data (if the previous application allowed an export).

Partition Tolerance

This criterion deals with the following question: assume that a publishing device p and a subscriber device s are in network A and a specific third-party provider device is in network B. There is no connection between A and B, e.g. due to a disrupted link. Does the provider scheme still maintain its functionality, i.e. does a data object get successfully delivered from p from s?

Required Infrastructure

The main reason why third-party providers are used for U2U communication in the first place, is data availability. A third-party provider device accepts data objects from publishing device and tries to deliver it to subscriber device. By doing so, publishing and subscriber device never have to communicate directly, i.e. they do not need to be on the same network simultaneously and still can trust in reliable data object delivery. For example, they can be connected to the Internet at disjunct times – the provider device persists the data object in between and acts as a relay.

CSP/FSP schemes typically achieve high data availability by high device availability[4] of the involved devices. Both provider schemes are based on one or multiple servers, i.e. **managed**

[4]The term "device availability" is to be distinguished from "content data availability". In this thesis, the shorter term "availability" is used if the context allows it. In case of possible ambiguity, the full terms are used.

devices with the aim of being available for other devices most of the time.[5] Structured P2P overlay networks achieve high data availability by replication of the data objects. Personal devices participate as peers in the overlay network and split the load of data object equally.

This criterion deals with the question whether one or multiple providers need to place infrastructure in form of managed devices, in order for the provider scheme to provide high data availability. If such infrastructure is required, this results in both acquisition and maintenance costs for the providers.

Resource strain on personal devices

This criterion complements previous criterion. It deals with the question how much resource strain on personal devices depending on the provider scheme – if personal devices are also provider devices (as is the case in structured P2P overlay networks), they have to store and deliver data objects by themselves and cannot shift this task to highly-available servers.

2.4.2 Centralized Service Provider

- **Application Metadata Privacy**: A CSP has the full view on all users of the same application. This omnipotence allows a CSP to exploit the users' PII in a more powerful way than it is possible for decentralized providers that only have a partial view on an application's users. For example, large scale data mining on the users' behaviour as a whole is easier with a full view. Additionally, a CSP is a more attractive for a *governmental attacker* [79] due to full access on an application's complete user data set. An infamous example for mass-collecting application metadata is *XKeyscore*, a formerly secret program of the U.S. National Security Agency (NSA) leaked by whistleblower Edward Snowden [27]. XKeyscore "*provides analysts with the capacity to mine content and metadata generated by e-mail, chat, and browsing activities through a global network of servers and internet access points*" [46].

- **Vendor Lock-In**: With a CSP, the application is inherently tied to its provider. If a user is for some reason unhappy with the provider, he cannot change without giving up the application altogether. This results in user groups having to find new consensus about applications. Messaging applications for mobile devices are one prominent example for this inflexibility: after Facebook had acquired the WhatsApp messenger [78], privacy concerns drove users looking for alternatives [110]. However, applications like Telegram [108], TextSecure [109] or Threema [111] are incompatible to each other and have non-congruent user bases. In order to reach as many contacts as possible, a user would have to install all these applications or stick to the most popular client, whereas this might change over time.

  Another drawback in centralization is the CSP being a single point of failure: In spite of all efforts towards high availability, the CSP can still be temporarily unavailable for some or all users, preventing access to the application. If the CSP decides to shut down its service (e.g. due to bankruptcy), the application becomes permanently unavailable for all users. In

[5]For comparison, [47] defines *managed devices* to be available at least 99% of the time. Google claims that their email service *Gmail* was available 99.978% of the time in 2013, which equals a downtime of less than two hours for the entire year. [67]

order to save his existing data, the user needs to be able to export his data set. Since data export is a feature that may or may not be provided by the CSP or by a third-party tool, this option is not always guaranteed.

- **Partition Tolerance**: Despite the high availability of a CSP's managed devices, accessing the CSP requires Internet connectivity for the users' devices. Therefore, if a publishing device p and a subscriber device s are inside the same network partition, but have no Internet connectivity, they cannot exchange any data objects via the application provided by the CSP. Therefore, the CSP provider scheme is not partition tolerant. A classic example for such a case is as follows: two co-workers are sitting next to each other on public transportation (e.g. train or airplane) and wish to exchange data, but have no Internet access due to a mobile connectivity dead zone. In this case, data exchange via a CSP is not possible.
- **Required Infrastructure**: For the application the CSP provides, the CSP needs to store all data of all users and is fully responsible for providing high data availability. A CSP achieves this by highly available managed device(s): depending on the number of users, a CSP needs to invest in large data centers with powerful, redundant hardware in order to provide a reliable application.
- **Resource strain on personal devices**: Due to a CSP's high availability, provider devices are reachable at arbitrary times for forwarding and delivering data objects. After successfully forwarding, a publisher can safely assume that the data object gets delivered to subscriber devices as soon as the CSP server and subscriber device are available for each other, i.e. the subscriber device has Internet connectivity. The publishing device does not maintain a list of subscriber devices, try to reach each subscriber device, etc. This keeps the resource strain on both types of personal devices low. Depending on the application, a server can also downsample application data (e.g. image quality) and decrease necessary data rates.

2.4.3 Federated Service Providers

- **Application Metadata Privacy**: Similar to a CSP, an FSP has the full view on all its customers with the same implication for their privacy. Unlike a CSP scheme, however, the users of the same application are scattered across different FSPs of their own choice. The number of users bound to a FSP (and therefore the ratio of the overall users) depends on the overall popularity of this specific FSP. In cases of extremely high popularity, an FSP can gain quasi-CSP status[6].

  Additionally, each user does not only have to trust his own FSP, but also the respective FSP of each contact. To see this, assume that user A uses FSP X and user B uses FSP Y. If A publishes a data object for B, this data object travels from A over X over Y to B. Even though there is no direct relationship between A and Y, Y learns that the data object it has to deliver to B comes from A.

  On the other hand, if the federation protocol is open (e.g. because it is free software or based on open standards), everybody can run their own FSP. FSPs administrated by

[6] In May 2015, Google claimed that its email service *Gmail* had over 900 million users [42].

non-profit organizations or even running on home servers, administrated by friends and family, are feasible. Users may trust such FSPs more than commercial providers and choose their provider accordingly.

- **Vendor Lock-In**: In contrast to a CSP, the application is not bound to a single provider. In order to use the application, each user can choose his own FSP and is free to switch to another one at any time. Exporting content data from the old FSP and importing content data to the new FSP may be a challenge.

  If only one FSPs becomes temporarily unavailable, only its customers are affected. Costumers of other service providers can still access the application and communicate with each other. If a service provider decides to shut down its service (e.g. due to bankruptcy), its customers can migrate to another service provider.

- **Partition Tolerance**: All publishing and subscriber devices require connectivity to their respective FSP server, they cannot transfer data objects directly. Therefore, the FSP provider scheme is not partition tolerant. Still, an FSP allows for more flexibility here than a CSP: for example, a company could setup an FSP on a local network and all the employees' devices can communicate with each other via this FSP without requiring Internet connectivity. However, as soon as these devices leave the local network or the FSP becomes otherwise unavailable for them, data object exchange is not possible anymore.

  Furthermore, if multiple FSP servers are involved in the delivery of a data object, these FSP servers need to be able to communicate with each other as well.

- **Required Infrastructure**: Similar to a CSP, an FSP needs to store all data of all its customers and is fully responsible for providing high data availability for the data objects it stores. An FSP achieves this by highly-available managed device(s): depending on the number of users, an FSP needs to invest in large data centers with powerful, redundant hardware in order to provide a reliable application.

- **Resource strain on personal devices**: Assuming all involved FSPs have high availability and reliable forwarding between different FSPs, provider devices are reachable at arbitrary times for forwarding and delivering data objects. After successfully forwarding, a publisher can safely assume that the data object gets delivered to subscriber devices as soon as the subscriber device and its respective FSP server are available for each other The publishing device does not maintain a list of subscriber devices, try to reach each subscriber device, etc. This keeps the resource strain on both types of personal devices low.

2.4.4 Structured P2P Overlay Networks

- **Application Metadata Privacy**: Even if a DHT does not require user identifiers in the sense that CSP/FSP do, all devices have NodeIDs which can be used for tracking implicit metadata. The responsible node for a specific data object can track store and lookup behaviour. Apart from peers that are responsible for storage, lookup behaviour may also be visible to peers on the lookup path. This strongly depends on the implementation of the DHT with regard to routing (iterative vs. recursive) and may also be obscured in a darknet style by source address rewriting [80].

This is the most notable property of structured P2P overlay networks with regard to privacy.

- **Vendor Lock-In**: In contrast to a CSP, the application is decoupled from specific providers. Once a user group has agreed on a specific application, they can keep using it as long as the overall P2P network is healthy enough: the application does not inevitably become unavailable, once one specific peer becomes unavailable. A structured P2P overlay network evenly distributes all data objects over the participating peers. If a peer leaves the overlay network, the remaining peers undertake its part for sharing data objects. Therefore, a structured P2P overlay network scales dynamically and exists as long as there are peers in the overlay network.

- **Partition Tolerance**: Typically, a Distributed Hash Table assumes that only one DHT instance needs to be maintained and that partitions do not exist. Assume that the peers in a DHT are located in two physical networks A and B and these networks A and B lose their connectivity due to a disrupted link. In this case, the peers in A (B) would regard the peers in B (A) as unavailable, which is a common and expected case in DHTs. Peers on the same network are still available, therefore two DHT partitions continue to work autonomously. As [96] puts it, "*a network partition, as seen from the perspective a single [peer], is identical to massive [peer] failures. Since [structured overlay networks] have been designed to cope with churn, they can self-manage in the presence of such partitions.*" In this sense, a DHT is indeed partition tolerant.

  However, such a split has strong implications: all data objects need to be replicated (shifted) to the new k closest peers, which results in a high effort and requires that the data objects are still locally available. Some data objects may be lost now for a partition, while others are not. The basic problem here is that DHT only takes one global instance into account and is inherently unaware of any other partitions. In order make a data object from one partition available to the other, there are two possibilities: Merging the two partitions or moving a peer which holds the data object to the other partition.

  Merging DHT partitions after a temporary network split is a non-trivial problem, as discussed in [55] and [96]: merging results again in a high effort for shifting data objects between the peers, with a trade-off between maintenance messages and time. Also, this procedures requires the two networks A and B to regain connectivity in the first place.

  The scenario of two DHT partitions working autonomously with mobile peers switching back and forth between networks A and B is even more difficult. Since all data objects are evenly distributed among all peers, with responsibility being randomly chosen by the NodeID, there is no DHT-based possibility for a specific mobile peer to carry a specific data object over to the other partition, unless it is by chance a responsible peer for said data object. Therefore, merging is the only feasible option for coordinating two partitions, which – as mentioned above – requires regained connectivity between A and B.

In summary, DHTs are not an ideal choice for scenarios where partition tolerance is important and two autonomous partitions have to be coordinated. All peers should have either Internet connectivity or be permanently on the same local network, analogous to the FSPs scheme.

- **Required Infrastructure**: A central aspect of structured P2P networks is the lack of necessity for a central server or other infrastructure. Instead via a dedicated server, the participating devices communicate in a self-organized way. Yet, for the overall performance of the P2P network, peers with high availability and fast connections are helpful.

- **Resource strain on personal devices**: By leveraging personal devices as peers, the need for CSPs and FSPs is eliminated. Instead of one provider storing all data objects for all its customers, each peer performs a fraction of the overall effort to keep the system running. DHTs are designed with fault tolerance and scalability in mind. Peer unavailabilities are expected and dealt with by replication. Each node only needs to maintain connections to a small subset of other nodes, but can reach any other node in the system by multi-hop routing. Lookups can be performed in logarithmic time, i.e. with $O(\log n)$ hops for n overall nodes in the system.

  However, for a given number of data objects, the responsibility load for each peer is inversely proportional to the overall number of active peers in the overlay network. With a large number of temporarily or permanently unavailable peers, the load for a single device can exhaust its resources.

  Replications can be expensive: depending on the size of a data object and the number of replicates k, replications can demand a high effort. Additionally, each replication has a certain latency depending on data object size and peers' network connectivity. If the peer availability fluctuates strongly, keeping all k replications on the k devices with the next-closest NodeIDs may be even unfeasible.

Other aspects

- **Sybil attacks:** By a sybil attack [28], an attacker creates a large number of virtual nodes in order to gain responsibility for large parts of the key number space – and thus data objects – or even a specific key. As a result, the attacker controls a specific value, i.e. can deny access to the respective data objects or learn about publisher and subscribers. Furthermore, he can disrupt signalling required for maintaining the DHT's functionality. With a sybil attack, a targeted attack on content data availability for a specific data object and application metadata privacy for specific IDs is possible.

- **Bootstrapping problem:** Unlike CSP and FSPs, there is no distinct and well-known provider to be contacted. Instead, each participating device needs to be integrated as a peer in the DHT. For doing so, it needs to contact an already existing peer and complete a bootstrapping procedure with the help of existing node. Finding an existing peer is a non-trivial problem and requires either probabilistic mechanisms (e.g. random address probing) or additional knowledge (external list of bootstrapping peers).

2.4.5 Summary

Table 2.1 summarizes the criteria comparison for the three third-party provider schemes Central Service Provider (CSP), Federated Service Providers (FSPs) and structured P2P overlay networks (DHT). The related work listed in Table 2.1 is presented in Section 2.5.

- **Application metadata not visible to third party:** Since all three provider schemes involve third-party providers, group members' application metadata is visible to them. However, there are qualitative and quantitative differences between the three: FSPs can be classified most privacy-friendly, since users can choose their providers and can consider whom they trust most. A DHT is less flexible in this regard since the users have no control which provider device stores their data. Still, each peer in a DHT has only a view on the PII for a part of the users. A CSP is the least privacy-friendly approach because it stores all data for every user of a given application – each user has to accept the CSP if they want to use the application.
- **No vendor lock-in:** Unlike for a CSP, there exist no vendor lock-ins for FSPs and structured P2P overlay networks: a customer of a specific FSP can switch to another FSP without having to give up the provided application. In structured P2P overlay networks, data objects are evenly distributed among the participating, therefore a concept such as vendors does not exist.
- **Partition tolerance:** CSPs and FSPs are not partition tolerant: all publishing and subscriber devices require connectivity to the CSP server/their respective FSP server. Even if publishing device p and subscriber device s are on the same network partition, the data object is always relayed via the third-party provider.

  A DHT only takes one global instance into account and is unaware of any unreachable partitions. DHT partitions maintain autonomous functionality after a network split. In this sense, a DHT is partition tolerant. However, the peers' responsibilities for data objects get re-distributed which results in high effort for shifting data objects between the peers. Merging two DHT partitions is associated with high maintenance costs as well. Transferring a specific data object via a specific mobile peer from one partition to another is not possible with the means of a DHT. Therefore, a DHT is partition tolerant, but coordinating two partitions is hard.
- **No infrastructure required:** Both a CSP and FSP require infrastructure in the form of managed devices, while a structured P2P overlay network does not require it once a peer has joined. Yet, for the overall performance of the P2P network, peers with high availability and fast connections are helpful. Additionally, the bootstrapping problem needs to be solved for new / re-joining peers.
- **Low resource strain on personal devices:** CSP/FSPs are highly-available server and take the storage and delivery load off personal devices. In a DHT, personal devices need to be provider devices themselves. This results in high resource strain on these devices, due to replications and routing etc.

Table 2.1: *Comparison of provider schemes*

	Centralized Service Provider (CSP)	Federated Service Providers (FSPs)	Structured P2P Overlay Networks (DHT)
Application metadata not visible to a third party	✗	✗	✗
No vendor lock-in	✗	✓	✓
Partition tolerance	✗	✗	(✓)[1]
No infrastructure required	✗	✗	✓
Low resource strain on personal devices	✓	✓	✗
Related work	*Persona* [4]	*Diaspora* [11]	*PeerSoN* [16]
	NOYB [50]	*SoNet* [95]	*LifeSocial.KOM* [45]
	FlyByNight [70]	*PrPl* [62]	*Safebook* [21]
		Friendica	*LotusNet* [1]
			DECENT [60] / *Cachet* [77]

[1] Eventual merging required, takes high effort.

2.5 Related Work

This section discusses related work about U2U communication with third-party providers. Most relevant works focus on online social networks (OSNs). OSNs are not clearly defined throughout the literature. Depending on the authors, OSNs are either equivalent to generic U2U communication as defined in this thesis, i.e. application-agnostic exchange of data objects within a closed group, or they follow a fix set of applications as offered by Facebook: profile pages, status updates, shared photo albums that can be commented, etc. In both cases, these are typical U2U communication use-cases and thus relevant.

2.5.1 Metadata and Privacy

Section 2.3.2 sketched how explicit and implicit metadata can affect user privacy. In this section, related work is presented which discusses the impact of metadata on privacy in more detail. Specifically, the aspects of application metadata shall be highlighted here.

The authors of [65] focus on the leakage of personally identifiable information (PII) in online social networks (OSNs), i.e. information that lets uniquely identify a specifc user. They present at the example of real-world OSNs how unique user identifiers, as well as user information like age and gender are exposed to ad networks. These types of information get exposed either by HTTP referer headers or external third-party apps which integrate into the OSN but run on external servers not affiliated with the OSN.

[49] gives an overview about privacy threats via application metadata and communication metadata in OSNs. The authors assume that each data object is encrypted and illegible to attackers. For application metadata, they discuss inferences from three categories: stored content, access control mechanisms and communication flows. For stored content, size of the encrypted data object, data structures such as the number of entries of linked data object in a list and modification histories (number of overwrites) are regarded. This information can be used to

learn about content types (text vs. image) or the frequencies of status updates or comments. Access control mechanisms can give away the number of recipients or contacts or even identities. Finally, communication flows deal with access patterns, such as the number of content requests or publishing behaviour. In order to gain information about the users from these categories, knowledge about access patterns, ciphertext, background knowledge or a combination thereof is required. However, the present countermeasures require mostly additional effort, e.g. by creating dummy entries and or generating noise traffic.

2.5.2 Privacy-Aware Online Social Networks

This section presents works about OSNs that target to improve the users' privacy over classic CSP-OSNs such as Facebook. With regard to privacy, the works mostly do not distinguish between application metadata privacy and content data security. Their common aim is to prevent an omnipotent CSP that can exploit either type of trust.

To this end, they introduce a decentralized component: either they leverage an existing CSP-OSN, but the client device does additional, decentralized work such as local encryption and decryption. Or the system of third-party providers is decentralized. Systems that either leverage CSP, are FSP-based or DHT-based are also listed in Table 2.1.

Two surveys papers give a good overview on privacy-aware OSNs: Schwittmann et al. [94] and Paul et. al [79].

Leveraging CSP-OSNs

These approaches build upon existing centralized OSNs such as Facebook and its advantages, such as leverage of the network's popularity and re-use of existing data. They add a decentralized component which increases the strain on personal devices. All presented approaches focus on encrypting or obfuscating content data. Implicit metadata such as the social graph are however still accessible for the CSP.

- *Persona* [4] uses attribute-based encryption (ABE) for end-to-end encryption. ABE allows for access control on the level of user attributes: first, each user defines different attributes and assigns an individual subset of these attributes to each of his contacts. This way, attribute-based groups of contacts emerge. Second, the user encrypts his data objects with regard to one or multiple attributes, logically linked. For example, by performing the "AND" operation on a set of attributes, only contacts that have all these attributes assigned can decrypt the data object. Persona was implemented by the authors as a Facebook application, i.e. leverages the users' existing social network. A Firefox extension performs encryption, signing and decryption decentrally in each user's local browser. Thus, Facebook does not have to be trusted with regard to content data confidentiality and content data integrity. On the other hand, Facebook has to be blindly trusted with regard to content data availability and application metadata privacy.

- *NOYB* [50] follows a similar idea as *Persona* by encrypting and decypting user data of regular Facebook accounts locally. However, NOYB does not encrypt user data in the classic sense of rendering it illegible. Instead, user data gets transformed into another seemingly legit value by exchanging user data pseudo-randomly with other NOYB users. The motivation here is to make the obfuscation process harder to detect for any attackers, specifically the CSP itself. Exchanges are stores in a public dictionary which must be made available by a trusted third party to all NOYB users.

- *FlyByNight* [70] is another approach to user data encryption in existing CSP-OSNs at the example of Facebook. Here, AES/El Gamal cryptography is used to encrypt messages and user data. The private keys are stored on a key repository. FlyByNight was implemented as a Facebook application. All cryptography was done in JavaScript, which resulted in poor performance according to the authors.

Federated Servers

These approaches are FSP-based with all the advantages and disadvantages discussed in Section 2.4.3.

- *Diaspora*[7] [11] is a DOSN based on federated servers. As a non-academic project, it gained major public attention in 2010 [31]. The founders explicitly marketed Diaspora as an alternative to centralized providers [101]. However, in 2014, over 70% of the approximately 380.000 users [94] used a server operated by the founders[8] as their provider. Diaspora's source code is freely available, thus everybody can run their own Diaspora server. While server-to-server communication is encrypted, all user data objects are stored unencrypted on the respective server and thus visible to the administrator. Besides the drawback of an FSP approach, each user additionally must blindly trust the server provider has to be blindly trusted with regard to content data confidentiality, content data integrity, content data availability and application metadata privacy.

- *SoNet* [95] is a DOSN also based on federated servers. Unlike Diaspora however, all data objects are stored encrypted on the FSPs: "*data objects exchanged between users are end-to-end secured by cryptographic operations. Servers therefore are not able to interpret them but merely forward them as opaque binary blobs.*" The result is a two-layered system: a federation protocol for forwarding the encrypted data objects between the servers and the encryption/decryption on clients. SoNet offers social graph obfuscation by adding a random identifier to each contact relationship. This identifier is stored on both involved FSPs and is used to forward messages to the correct recipient without exposing cleartext usernames to the other FSP. This approach only works as long as both users use different FSPs and the two FSPs do not cooperate with each other.

- *PrPl* [62] (short for *private-public*) is also based on federated servers, but introduces an abstraction layer (the *Personal-Cloud Butler*) which lets the users store their data using different storage vendors. Client applications also communicate via an abstraction layer (the *Pocket Butler*) which handles unified authentication, authorization and lookup requests.

[7]https://diasporafoundation.org
[8]http://joindiaspora.com

- *Friendica*[9] is another open-source project for FSP-based DOSNs that is very similar to Diaspora.

Structured P2P Overlay Networks

These approaches leverage a DHT with all the advantages and disadvantages discussed in Section 2.4.4.

- *PeerSoN* [16] is a DOSN with a two-step approach. First, a DHT is used as a lookup service which is used to find the devices which hold a specific data object, as well as locator information (e.g. IP addresses) of these devices. Second, the data object can be directly requested from the found devices with this information. This two-step approach eliminates the replication of large data objects in the DHT. This results in less resource strain on personal devices at the cost of worse data availability. However, the first step still exposes application metadata privacy to third parties: requests for a data object expose users' interest in that data object to the same peer which holds the mapping between said data object and the devices that store it.

- *LifeSocial.KOM* [45] is a DOSN which offers a set of applications similar to Facebook. It follows a similar approach to *PeerSoN*. Unlike *PeerSoN* however, all data objects are fully stored in a DHT. Before a data object gets stored in the DHT, each data object is encrypted symmetrically, whereas the symmetric key gets encrypted with each read-enabled user's public key. The list of symmetric keys, as well as the encrypted data object gets signed with the creator's public key, thus each data object is linkable to its creator if the matching public key is identified. New applications can be realized within LifeSocial.KOM in form of plugins. *LifeSocial.KOM* is agnostic of the used DHT and proposes to leverage one of the existing off-the-shelf DHT implementations such as FreePastry [87] or OpenDHT [85]. This impacts application metadata privacy as discussed in Section 2.4.4. Since all data objects are fully stored in the DHT, regardless of their number and size, replications can be expensive. This can result in a high strain on personal devices.

- *Safebook* [21] is a DHT-based DOSN with additional privacy features. The device of user *A* connects directly to the devices of *A*'s trusted contacts. These contact devices form an "inner shell" around user *A*. *A* stores his personal data only on devices of that inner shell. The devices of the inner shell in turn connect with the devices of their trusted contacts, which results in another (friend-of-a-friend) shell around *A*'s inner shell, and so on. The sum of all shells around *A* is called *A*'s *matryoshka*. All devices on the outermost shell register themselves as entrypoints for *A* in the DHT. If any other device wants to send a message to *A*, it first looks up the list of entrypoints in the DHT and sends the message to one of *A*'s entrypoint. From the entrypoint, the message is routed through *A*'s matryoshka, where each hop is a trusted link. This approach has two notable advantages over a regular DHT: first, it prevents that *A*'s personal data is stored in the DHT, since it is only stored on *A*'s inner shell, which is more trusted than a random third party. Second, requests for data

[9] http://friendica.com

retrievals by *A* and towards *A* are anonymized, similar to friend-to-friend networks. Still, Safebook has to deal with a DHT's problems: The DHT peer which is responsible for *A*'s list of entrypoints could deny access to that list. The same peer also learns about interest of other peers in *A*. However, Sybil attacks are prevented by a *trusted identification service* (TIS) that Safebook provides. The TIS is another third party that needs to be trusted.

- *LotusNet* [1] is a DOSN which is based on the DHT *Likir* [2], a previous work of the authors. Likir is an identity-based extension of *Kademlia* [71] which prevents peers from giving themselves a large number of arbitrary NodeIDs. This way, Likir prevents Sybil and other DHT-specific attacks. NodeIDs are tracked by a certification service, similar to *Safebook*'s trusted identification service.

- *DECENT* [60] is a DOSN where all data objects are fully stored in a DHT. Similar to *Persona*, DECENT uses attributed-based encryption (ABE) for ensuring content data confidentiality and content data integrity. Additionally, it supports efficient access rights revocation based on the EASiER mechanism [59], a previous work of the authors. Still, application metadata privacy and possibly high resource strain on personal devices is a problem due to the leverage of off-the-shelf DHT implementations like FreePastry [87] and Kademlia [71]. In order to reduce high computational efforts for encryption and decryption, the successor *Cachet* [77] was designed. Here, data objects are additionally stored unencrypted on trusted contact devices besides the DHT. The *DECENT* approach is used as a fallback in case of contact devices being unavailable.

Other approaches

Mantle [35] follows a straightforward approach here where each user runs the Mantle application on a client device. Additionally, each user maintains his own storage server. Each storage server holds a public profile of the respective user, including a profile picture, status and public key. Two contacts need to exchange the address to each other's storage service out-of-band. The Mantle application on the client device of user *A* can then download the public key of user *B* from *B*'s storage server. Data objects from *A* for *B* get encrypted with *B*'s public key and uploaded to *A*'s storage server. *B* in turn downloads the data object from *A*'s storage server and decrypts the message locally on his client device with the Mantle application which holds the private key. Application metadata is thus visible to the publisher's (here: *A*) storage server provider.

Vegas [30] focuses on keeping the social graph secret not only from providers, but also from contacts. Unlike other OSNs, it is not possible to crawl a user's contact list to search for common contacts. Vegas stores all data objects encrypted with random file names on public storage servers that each user can freely choose. It is also possible to storage data objects for different contacts refer on different servers. The location and file names of the data objects must be propagated via a so-called *exchanger* to the subscribers. An exchanger is a side channel which allows for asynchronous, mailbox-like communication (e.g. email or SMS). Thus, application metadata visibility depends on the used data storage providers and exchangers. Assuming that classic communication like email is used, most of the application metadata privacy problem is merely shifted to the exchanger level.

There are other approaches for DOSNs where all data user is stored on devices that the respective user generally trusts. These are more similar to User-Centric Networking. Thus they will be discussed in the next chapter, namely Section 3.7.

2.5.3 Friend-to-Friend Networks

Friend-to-friend networks or *darknets* are another approach to protect their users' privacy during communication. Here, each peer only communicates with trusted peers. The approach is based on the assumption for each peer that his trusted peers never give away information about the network and forward every message unmodified and correctly according to the network protocol. Typically, a darknet targets at anonymity for senders and recipients, as well as unlinkability of messages to senders and recipients.

Examples here are multi-hop darknets which focus on anonymous file sharing and circumventing censorship, such as *Turtle* [81], GNUNet [10] or Freenet [18]. More recent work deals with efficient routing in darknets [86].

Darknets cover a different scenario than U2U communication. In U2U communication, not only the content of each data object, but also the identity of the publisher is relevant. Thus, sender anonymity is not an aspect of U2U communication. Instead, linkability is an explicit property of a data object (see Section 2.1). A darknet can still be used to implement U2U communication, but since a darknet's feature of sender anonymity is not required in the U2U scenario, any extra overhead (and thus e.g. possible performance impacts) to achieve this central feature is unneeded extra work for enabling U2U communication.

2.6 Conclusion

This chapter presented the basics for user-to-user (U2U) communication. In Section 2.1, a model for U2U communication scenario was presented, which is the foundation for all considerations in this thesis: a data object shall be shared between the members of a closed group. The data object is both confidential and linkable to its publisher. Each closed group is defined on per-data-object basis.

For U2U communication, typically *third-party providers* are used for storing data objects and making them permanently available to the closed group. Section 2.2 presented three *third-party provider schemes* that are typically used: Centralized Service Provider, Federated Service Providers and structured P2P overlay networks.

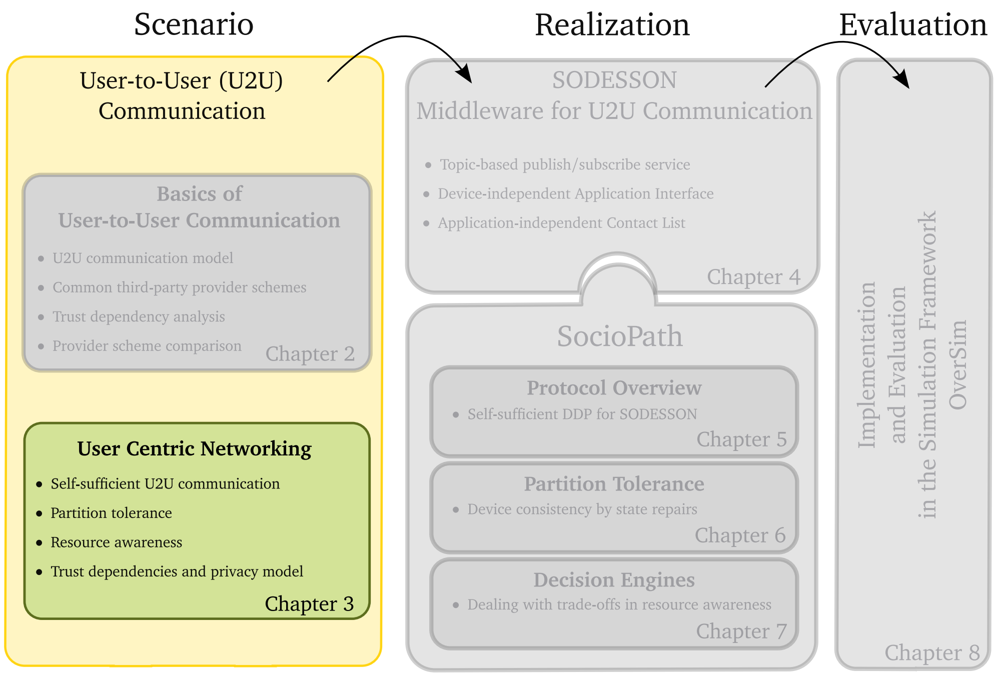
In Section 2.3, four security and privacy goals for data objects were discussed. For each goal, trust dependencies were identified from the closed group members to third-party providers in general as well as between the members themselves.

As one result of the trust dependency identifications, all closed group members must blindly trust a third-party provider with regard to application metadata privacy. In Section 2.4, the three presented third-party provider schemes were each analyzed in this regard. Additionally, the schemes were analyzed with regard to four other aspects: vendor lock-in, partition tolererance, required infrastructure and resource strain on personal device. These aspects will be revisited in Chapter 3 for comparing User-Centric Networking.

Finally, Section 2.5 discussed related work. First, publications that focus on the impact of application metadata on user privacy were presented. Second, approaches to privacy-aware U2U communication with focus on online social networks were presented. Each approach leverages one of the three presented third-party provider schemes, and thus have their inherent disadvantages.

Chapter 3

User-Centric Networking



Scenario
Realization
Evaluation
User-to-User (U2U) Communication
Basics of User-to-User Communication
U2U communication model
Common third-party provider schemes
Trust dependency analysis
Provider scheme comparison
Chapter 2
User Centric Networking
Self-sufficient U2U communication
Partition tolerance
Resource awareness
Trust dependencies and privacy model
Chapter 3
SODESSON Middleware for U2U Communication
Topic-based publish/subscribe service
Device-independent Application Interface
Application-independent Contact List
Chapter 4
SocioPath
Protocol Overview
Self-sufficient DDP for SODESSON
Chapter 5
Partition Tolerance
Device consistency by state repairs
Chapter 6
Decision Engines
Dealing with trade-offs in resource awareness
Chapter 7
Implementation and Evaluation in the Simulation Framework OverSim
Chapter 8

Figure 3.1: *Placement of Chapter 3 in the big picture*

In Section 2.4, the drawbacks of U2U communication via common third-party provider schemes were evaluated: first, it was discussed that the members of a closed group have a blind trust dependency to third-party providers with regard to application metadata privacy and content data availability.

Second, neither of the three compared third-party provider schemes can properly handle partitions: CSP/FSP schemes are not partition tolerant at all, DHTs are designed with one global instance, not the coordination of autonomous partitions in mind.

Third, in order to provide high data availability, a third-party provider scheme either requires highly-available infrastructure (CSP/FSP) or high resource strain on personal devices due to multi-hop routing and – if needed – replications (DHT).

This chapter discusses User-Centric Networking as an alternative to user-to-user communication via third-party providers. User-Centric Networking aims at overcoming the three mentioned drawbacks of third-party providers:

- **Increased application metadata privacy by self-sufficiency**: All U2U communication is performed without any third-party providers. Instead, the personal devices of a closed group's members are not only used to publish / consume a data object via an application, but also act as provider devices for storing and delivering data objects to subscriber devices. Access rights, i.e. allowed publishers and allowed subscribers, are controlled by the topic owner. Each published data object has to travel via a topic owner's device first which decides whether the data object comes from an allowed publisher and to which subscribers the data object may be delivered.

  Thus, the topic owner – who has defined the allowed publishers and allowed in the first place – plays a central role: he maintains the access rights and is responsible how a published data object gets distributed inside the closed group. In any case, non-members of the closed group or any other third partys are excluded – which ensures the privacy of the closed group with regard to third parties. Additionally, there are also some freedoms of degree in what manner data objects get distributed in the closed group with different effects on the intra-group privacy. For example, the topic owner may or may not ask subscriber A to deliver a data object to subscriber B and thus may or may not expose A's and B's subscribership to each other.

  This type of U2U communication is called *self-sufficient U2U communication* or **self-sufficiency**. By completely removing third-party providers from U2U communication, neither content data nor application metadata is visible to third parties anymore. Adversaries can only participate if the topic owner made them allowed publishers and/or allowed subscribers. In that case, they are technically not a third party anymore, but valid members of closed groups. Self-sufficiency is discussed in detail in Section 3.1.

- **Partition tolerance**: Data delivery in a UCN relies on personal devices. Personal devices can lack Internet connectivity at times, be moved from one local network to the other or perform opportunistic networking via ad-hoc or delay-tolerant networks with other devices. Such diverse device availability situations can result in disjunct partitions, where some devices are able to communicate with each other while other devices are not available.

  Devices within the same partition should be able to exchange data objects, regardless of any devices of the same or other users outside that partition. For example, this would enable U2U communication for two users on an airplane, without Internet connectivity, but with their local devices communicating directly via Bluetooth or WiFi Direct. This local ad-hoc network forms an autonomous partition.

  Data objects should also be transferable from one partition to the other. If two partitions merge (e.g. a disrupted link between two networks becomes functional again), the previously published data objects can now be delivered to the remaining subscriber devices. Another possibility is a mobile topic owner device which carries a data object from network $\mathcal{A}$ to network $\mathcal{B}$, comparable to a delay-tolerant pocket switched network. Partition tolerance in User-Centric Networking is discussed in more detail in Section 3.3.

- **Resource-aware leverage of personal devices**: As discussed in Chapter 2, third-party provider schemes achieve high data availability either by highly-available infrastructure (CSP/FSP) or by leveraging personal devices and cope churn with replication on multiple devices (DHT). User-Centric Networking aims at a middle road: self-sufficiency requires personal devices as provider devices, but a distinction between *weaker devices* (less resources) and *stronger devices* (more resources) is made. Many users have multiple devices at their disposal from smartwatches over smartphones up to PCs and small home servers (e.g. network-attached storages). These device types are heterogenous in terms of resourcefulness and availability. User-Centric Networking takes these differences into account when coordinating access control and data delivery across the devices. As a result, the combined resources of all group members' devices can and should be used as provider devices. If possible, provider tasks shall be mainly performed by stronger devices. Resource-awareness is discussed in more detail in Section 3.2.3.

Table 3.1: *Comparison of User-Centric Networking with third-party provider schemes*

	CSP	FSPs	DHT	**User-Centric Networking**
Application metadata not visible to a third party	✗	✗	✗	✓
No vendor lock-in	✗	✓	✓	✓
Partition tolerance	✗	✗	(✓)[1]	(✓)[2]
No infrastructure required	✗	✗	✓	✓
Low resource strain on personal devices	✓	✓	✗	(✗)[3]

[1] Eventual merging required, takes high effort.
[2] Partition must contain topic owner device to be fully functional.
[3] Resource awareness aims to reduce strain on weak devices.

3.1 Self-sufficient U2U communication

Self-sufficient U2U communication (self-sufficiency) is a novel provider scheme for U2U communication, firstly introduced in the SocioPath publication [51]. As one of the three design goals for User-Centric Networking, it is based on User-Centric Networks (UCNs), which are defined as follows. Subsequently, general data object delivery in UCNs is explained.

3.1.1 User-Centric Network (UCN)

In self-sufficiency, each published data object is associated to a data-object-specific User-Centric Network (UCN). A UCN contains only devices of those users that are actually allowed to access the published data object: the publisher, the topic owner and the allowed subscribers. Thus, a

UCN is a direct mapping of the data object's closed group. The associated data object is to be delivered to the subscriber devices with help of the devices in UCN only. Since each UCN is bound to a specific data object, the existence of a UCN is ephemeral for the time when the data object is to be distributed. For different data objects, a user or device can be part of multiple UCNs at the same time, each for one specific data object.

Formal definition

A *User-Centric Network (UCN)* is a 6-tuple $\mathcal{N}_{\delta_t} := \{t,\ \delta_t,\ TO_t,\ P_{\delta_t}, \mathbb{S}_t, \mathbb{D}_{\delta_t}\}$ comprising of:

(1) t: **topic**

(2) δ_t: **data object** with topic t

(3) TO_t: **topic owner (TO)** of t (single user)

(4) P_{δ_t}: **publisher** of data object δ_t (single user)

(5) $\mathbb{S}_t$: **subscribers** of topic t (set of users)

(6) $\mathbb{D}_{\delta_t}$: **device pool** for data object δ_t (set of devices)

Let the set of all users in $\mathcal{N}_{\delta_t}$ be $\mathbb{U}_{\delta_t} := \{TO_t,\ P_{\delta_t}, \mathbb{S}_t\}$ Let user $X \in \mathbb{U}_{\delta_t}$ and all devices under user X's control be $\mathbb{D}_X$. Then, $\mathbb{D}_{\delta_t} := \bigcup_{X \in \mathbb{U}_{\delta_t}} \mathbb{D}_X$. Every device is assumed to be under exactly one user's control, i.e. $\mathbb{D}_A \cap \mathbb{D}_B = \emptyset \mid \forall A, B \in \mathbb{U}_t, A \neq B$.

The discussed entities of a UCN are summarized in Table 3.2.

Purpose of a UCN

As given by the tuple, each UCN $\mathcal{N}_{\delta_t}$ is associated to one specific data object δ_t with topic t. δ_t shall be delivered to $\mathbb{D}_{\delta_t}$, i.e. all devices of the publisher, all devices of the topic owner and all devices of all subscribers of t. Since only the topic owner knows all subscribers, the data object has to travel across his devices first. On each device $d \in \mathbb{D}_{\delta_t}$, δ_t can be consumed by the respective user via an application associated to t, running on that device.

Only devices in $\mathbb{D}_{\delta_t}$ act as provider devices, i.e. are leveraged to deliver δ_t to $\{\mathbb{D}_{\mathbb{S}_t} \cup \mathbb{D}_{P_{\delta_t}}\}$. Third partys are not involved.

Role of the topic owner and topic owner devices

Independent from a specific UCN, each user A has a set of **contacts** $\mathbb{C}_A$. Contacts are bidirectional, i.e. $A \in \mathbb{C}_B \Leftrightarrow B \in \mathbb{C}_A$. Let $A \in \mathbb{C}_A$.

For each topic t, the associated TO_t defines the set of allowed publishers $\mathbb{P}_t \subseteq \mathbb{C}_{TO_t}$ from his contacts. Likewise, TO_t defines the set of allowed subscribers $\mathbb{A}_t \subseteq \mathbb{C}_{TO_t}$ from his contacts. Thus, the audience of a published data object is limited by the topic owner only – a publisher himself cannot individually constrain which users shall receive the published data object.

For each UCN associated to a data object with topic t, i.e. $\mathcal{N}_{\delta_t}, \mathcal{N}_{\epsilon_t}, \mathcal{N}_{\zeta_t}, \ldots$, it is $P \in \mathbb{P}_t$ and $\mathbb{S}_t \subseteq \mathbb{A}_t$ (with P being $P_{\delta_t}, P_{\epsilon_t}, P_{\zeta_t}, \ldots$ respectively).

$\mathbb{D}_{TO_t}$, i.e. the devices of TO_t, are the only devices which store the sets of $\mathbb{P}_t$, $\mathbb{A}_t$ and $\mathbb{S}_t$. Therefore, TO devices are the only devices that can perform access control – each published data object with topic t needs to be forwarded to a TO device first, since only the TO devices know which users are subscribers. Before delivering a data object δ_t to the subscriber devices, the TO device verifies that δ_t was published by an allowed publisher, i.e. $P_{\delta_t} \in \mathbb{P}_t$.

There are several reasons for this design decision:

- **Centrality:** Each member of a closed group is a contact of the topic owner. Therefore, the topic owner plays a central role in each UCN. Assume that each device of user A knows all devices of all of A's contacts. For every UCN where A can participate (since A and the respective TO are contacts), a can be a publishing device, as it knows how to reach each TO device. In analogous way, each TO device knows how to reach every subscriber device, since subscribers are again contacts of the TO. Therefore, this TO centrality achieves that the effort in device maintenance grows with the number of contacts, but not with the number of UCNs a user participates in. As a downside, the topic owner is a single point of failure in each UCN.

- **Privacy:** If only topic owner devices deliver data objects to subscriber devices, no group member other than the TO can learn which other users are also subscribers. Even inside a UCN, a higher privacy level is therefore possible. A privacy model is discussed in Section 3.6.

- **Security:** The topic owner defines the sets of allowed publishers and allowed subscribers. If only his devices hold this information, no other group member can manipulate the access rights, add third parties to the closed group etc.

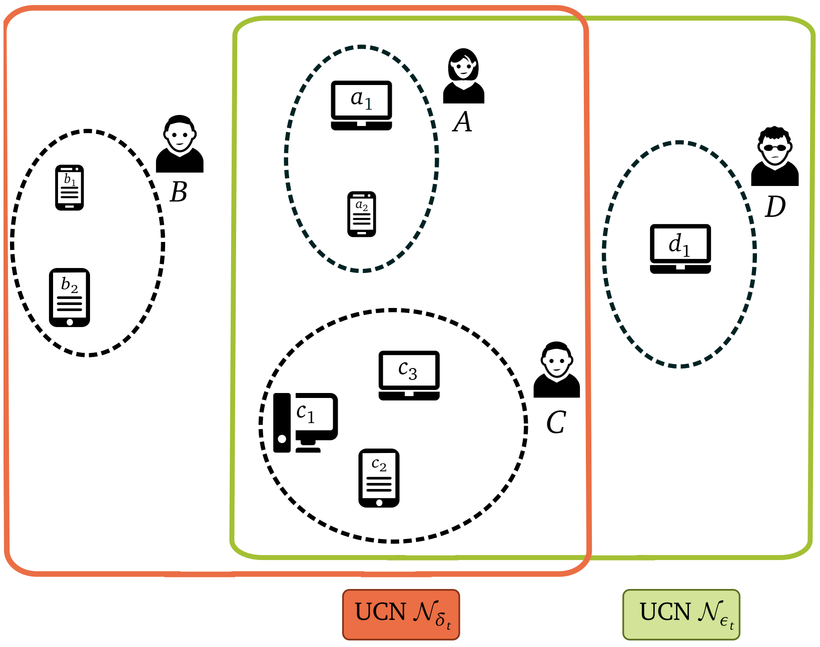
The disadvantage of the TO's centrality is that data object delivery completely depends on the TO's devices. A data object δ_t cannot be delivered to $\mathbb{D}_{\mathbb{S}_t}$, if no device in $\mathbb{D}_{TO_t}$ is available. The delivery process is stalled until at least one of the TO's devices becomes available again. In that case, TO devices need to synchronize with the devices of allowed publishers and learn whether new data objects with the respective topic have been published.

Example

For topic t, user A is the TO, i.e. $TO_t = A$. A's contacts are $\mathbb{C}_A = \{A, B, C, D\}$. A has defined the allowed publishers $\mathbb{P}_t = \{A, B, D\}$ and the allowed subscribers $\mathbb{A}_t = \{A, C, D\}$, each are subsets from $\mathbb{C}_A$. Of the allowed subscribers, only C has subscribed to t: $\mathbb{S}_t = \{C\}$.

Allowed publisher B publishes δ_t, thus the UCN $\mathcal{N}_{\delta_t} = \{t,\ \delta_t,\ A,\ B,\ \mathbb{S}_t,\ \mathbb{D}_{\delta_t}\}$ emerges. Note that user D is neither publisher of δ_t nor subscriber for t and therefore not a member of $\mathcal{N}_{\delta_t}$, despite D being both an allowed publisher and allowed subscriber. Data object δ_t will never be forwarded to D's device d_1.

Allowed publisher D publishes ϵ_t, thus the UCN $\mathcal{N}_{\epsilon_t} = \{t,\ \epsilon_t,\ A,\ D,\ \mathbb{A}_t,\ \mathbb{S}_t,\ \mathbb{D}_{\epsilon_t}\}$ is defined. Here, user B is excluded from $\mathcal{N}_{\epsilon_t}$, because B is neither the publisher of ϵ_t nor an allowed subscriber of t. Data object ϵ_t will never be forwarded to B's devices b_1 and b_2.



$\mathbb{C}_A = \{A, B, C, D\}$
$TO_t = A$
$\mathbb{A}_t = \{A, C, D\}$
$\mathbb{S}_t = \{C\}$
$\mathbb{P}_t = \{A, B, D\}$

$P_{\delta_t} = B$
$\mathbb{D}_{\delta_t} = \{a_1, a_2, b_1, b_2, c_1, c_2, c_3\}$

$P_{\epsilon_t} = D$
$\mathbb{D}_{\epsilon_t} = \{a_1, a_2, c_1, c_2, c_3, d_1\}$

Figure 3.2: *Example of two UCNs: $\mathcal{N}_{\delta_t}$ and $\mathcal{N}_{\epsilon_t}$ for data objects δ_t and ϵ_t respectively.*

As can be seen from these examples, an allowed publisher is only part of a UCN if he is the publisher of the respective data object. An allowed subscriber is only part of a UCN is a subscriber. The latter is only the case, if the allowed subscriber has previously decided that he is interested in the given topic and thus decided to actually subscribe.

Both example UCNs $\mathcal{N}_{\delta_t}$ and $\mathcal{N}_{\epsilon_t}$ for data objects δ_t and ϵ_t respectively are displayed in Figure 3.2.

3.1.2 Data object delivery

Figure 3.3 displays the basic workflow of self-sufficiency. The basic steps for delivering a data object δ_t from its publisher to a subscriber are as follows. For now, assume that there is only one device per user. Additional devices are discussed afterwards.

(1) The TO defines $\mathbb{P}_t$ and $\mathbb{A}_t$ on his TO device.

(2) An allowed subscriber S uses his device to subscribe to t.

(3) This subscription request is sent from the **subscriber device** to the TO device.

(4) The TO device verifies that $S \in \mathbb{A}_t$ and updates $\mathbb{S}_t$ accordingly.

Table 3.2: *Overview on the entities in a UCN*

Symbol	Entity	Comment
$\mathcal{N}_{\delta_t}$	UCN for data object δ_t with topic t	
TO_t	Topic owner of topic t	
$\mathbb{A}_t$	Allowed subscribers for topic t	
$\mathbb{S}_t$	Subscribers of topic t	$\mathbb{S}_t \subseteq \mathbb{A}_t$
$\mathbb{P}_t$	Allowed publishers for topic t	
P_{δ_t}	Publisher of data topic δ_t	$P_{\delta_t} \in \mathbb{P}_t$
$\mathbb{U}_{\delta_t}$	All users in UCN $\mathcal{N}_{\delta_t}$	$\mathbb{U}_{\delta_t} := \{TO_t, P_{\delta_t}, \mathbb{S}_t\}$
$\mathbb{D}_X$	Devices of User X	
$\mathbb{D}_{\mathbb{S}_t}$	Devices of subscribers of topic t	$\mathbb{D}_{\mathbb{S}_t} = \bigcup_{X \in \mathbb{S}_t} \mathbb{D}_X$
$\mathbb{D}_{\delta_t}$	Device pool of UCN $\mathcal{N}_{\delta_t}$	$\mathbb{D}_{\delta_t} = \{\mathbb{D}_{TO_t} \cup \mathbb{D}_{P_{\delta_t}} \cup \mathbb{D}_{\mathbb{S}_t}\}$

(5) Allowed publisher P uses his device to publish a new data object δ_t.

(6) This data object is forwarded from the **publishing device** to the TO device.

(7) The TO device verifies that $P \in \mathbb{P}_t$ before δ_t gets delivered to any subscriber device, i.e. it is ensured that the data object comes from an allowed publisher.

(8) The TO device knows that $S \in \mathbb{S}_t$ and therefore the TO device delivers the δ_t to the subscriber device of S.

(9) Subscriber S can now consume δ_t on his subscriber device.

Thus, any published data object δ_t has to pass the TO device first, since the TO device is the only device which performs access control for the publisher (i.e. it can decide whether the publisher is actually allowed to publish data objects for topic t) and knows the subscribers.

Note that in cases where the publisher is also the TO, the publishing device is also a TO device. Here, the steps (6) and (7) are omitted. Analogously, a subscriber device is a TO device if the TO is a subscriber. Here, step (8) is omitted.

Additional devices

If additional devices exist in the device pool for topic t, the described basic steps need to be extended by additional steps. Each respective device mentioned above shall be named *first device* here. This does not indicate any hierarchy, but merely specify the device which is taken for user interaction here.

- **Additional topic owner devices:** The first TO device needs to propagate the TO's definitions for $\mathbb{P}_t$ and $\mathbb{A}_t$ to all other TO devices. Likewise, new subscriptions, i.e. changes in $\mathbb{S}_t$ need to be propagated to all other TO devices.

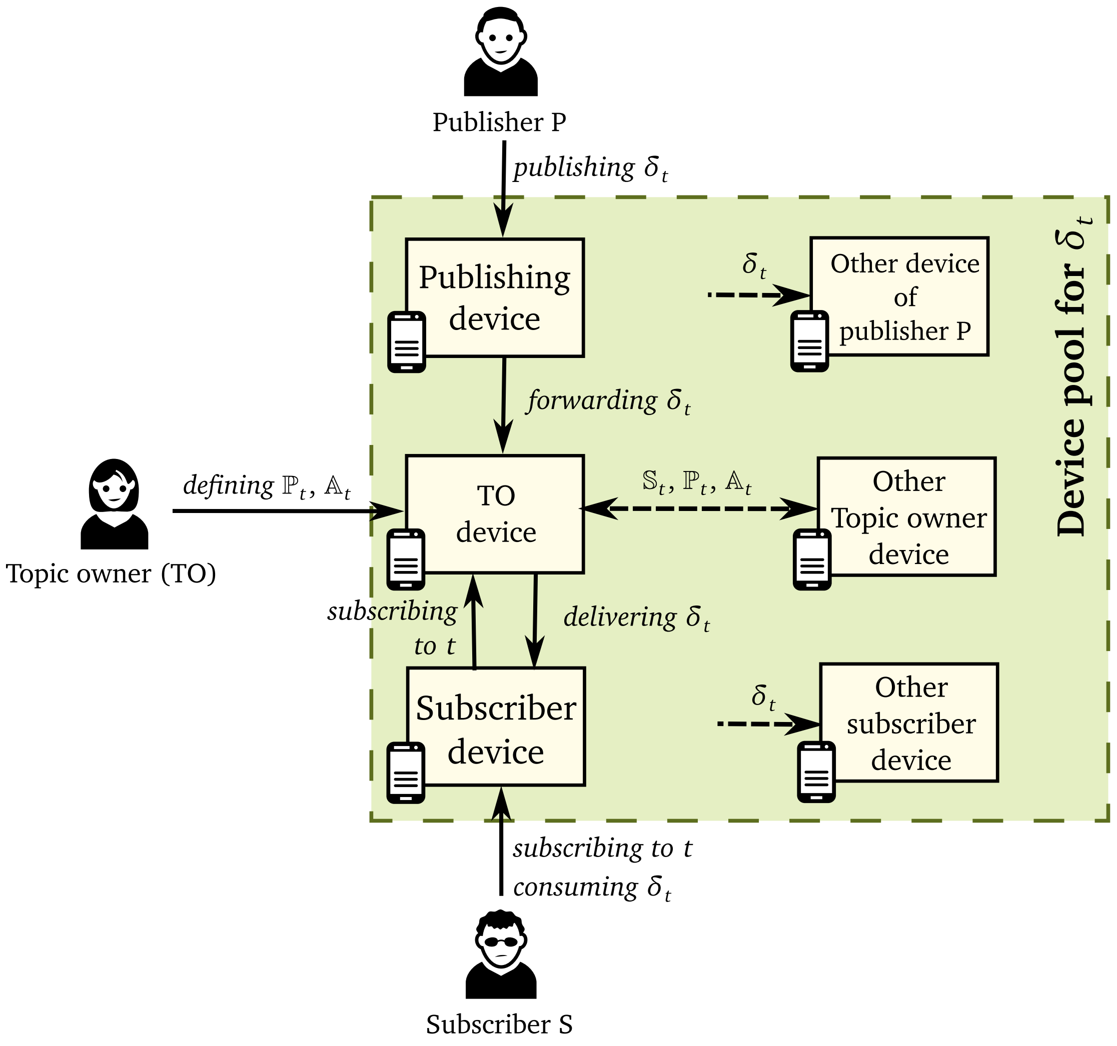


Figure 3.3: *Self-sufficient U2U communication*

- **Additional devices of publisher P:** The data object δ_t published on P's first device shall also be delivered to all other devices of P (but no other allowed publishers, unless they are subscribers).

- **Additional devices of subscribers:** The data object δ_t delivered to the first subscriber device of S shall also be delivered to all other subscriber devices – devices of S and each other subscribers $X \in \mathbb{S}_t$.

So far it is unspecified which **forwarding policy** is used, i.e. which device(s) forward / deliver δ_t to any additional devices. The only constraint in self-sufficiency is that this problem must be solved with devices from the device pool $\mathbb{D}_t$ only. Examples for different forwarding policies are shown in Section 3.4.1.

3.2 Available Resources in a UCN

Self-sufficient U2U communication fully relies on the device pool in a given UCN, i.e. on the personal devices of the respective users. Personal devices are heterogeneous with regard to their resources. A device's resources have the following two characteristics: **device availability** and **device capability**.

3.2.1 Device availability

Device availability describes the fundamental ability of a device to communicate with another device at a given time. This aspect is relevant for devices that have a central role in forwarding data objects. For example, if the only device of TO_t is unavailable most of the time, published data objects with topic t cannot be delivered to the subscriber devices, as they have to be forwarded to the TO device first.

The device availability $\alpha(d_1, d_2, t_0)$ is defined between two devices d_1 and d_2 for a specific time t_0. d_1 is **available for** d_2 (and vice-versa) if d_1 and d_2 can communicate over an underlying network at t_0.

Examples for an underlying network are:

- both devices have Internet connectivity
- both devices are on the same local network
- both devices have formed an ad-hoc network

If two devices d_1 and d_2 are not on same network at the same time, they are in different **partitions** and are **unavailable** for each other during that time.

3.2.2 Device capability

Device capability defines the effort a device can and must make, based on the number and size of data objects. Effort means here

- storing data objects
- forwarding data objects to TO devices
- delivering data objects to subscriber devices
- receiving data objects from other devices

These efforts are bound to resource costs for the affected devices. Example resources are CPU load, memory, battery power, storage space, monetary connectivity costs and connectivity bandwidth.

3.2.3 Example Device Classes

Based on the resources types, three different device classes are now defined in order to demonstrate device heterogeneity. These classes are also blueprints for simulated devices in later chapters. They are described as follows and summarized in Table 3.3.

- *Smartphone*: assumed to have *high availability*. It is switched on most of the time and has at least a very slow data connection which can be used for U2U communication over the Internet. However, it is also assumed to be a mobile device with a metered cellular connection where every byte counts against the data volume of the user's mobile contract, i.e. costs money. Since this poses a critical bottleneck for uploading and downloading data objects, a smartphone is assumed to have *low capability*.
- *Laptop*: assumed to have *low availability*. When it is not actively used by the user, it gets snapped shut, falling into sleep mode. Even if it is a mobile device, it is assumed to use WiFi connectivity only which is not available everywhere. However, if the laptop has WiFi connectivity, this WiFi is assumed to be fast and cheap or even free for the user. A laptop also has a strong CPU and today's hard drives provider ample storage. Therefore it is assumed to have *high capability*.
- *Home server*: assumed to have *high availability*. As a server-type device, it almost always switched on to be available for requests. It can be connected 24/7 to the Internet via a landline DSL connection and thus provide connectivity almost without a break. It is also connected to a power socket, has a large harddisk attached to it and the DSL connection is fast and not metered, but provided by a flatrate contract. Therefore, a home server is assumed to have *high capability*.

Table 3.3: *Device classes in User-Centric Networking*

Device class	High availability	High capability
Smartphone	✓	✗
Laptop	✗	✓
Home Server	✓	✓

3.3 Partition tolerance

As mentioned earlier, personal devices can undergo changes in their availability: they can lack Internet connectivity at times, be moved from one local network to the other or perform opportunistic networking via ad-hoc or delay-tolerant networks with other devices. Such diverse device availability situations can result in disjunct partitions, where some devices are able to communicate with each other while others are not.

If the device pool $\mathbb{D}_t$ of UCN $\mathcal{N}_t$ is split among different partitions, each partition should work autonomously as long as the workflow described in Section 3.1.2 is still possible, i.e. the required devices are within the same partition.

Assume two disjunct networks (partitions) $\mathcal{A}$ and $\mathcal{B}$ and a topic t. Partition $\mathcal{A}$ contains three devices: one of an allowed publisher $P \in \mathbb{P}_t$, one of TO_t and one of a subscriber $S \in \mathbb{S}_t$. Partition $\mathcal{B}$ contains three devices, each one of the very same users. Within either partition, data objects can be published by the allowed publisher and get delivered to the subscriber device via the topic owner device. Each partition works autonomously, as long as it contains a topic owner device.

As discussed in Section 3.1.1, each data object δ_t shall be delivered $\mathbb{D}_{\mathbb{S}_t} \cup \mathbb{D}_P$, i.e. to all devices of all subscribers and to all devices of the respective publisher. Therefore, if δ_t was published in partition $\mathcal{A}$, it shall eventually be delivered to the devices in partition $\mathcal{B}$. There are different possibilities to achieve this: Two partitions can merge (e.g. a disrupted link between two partitions becomes functional again), therefore bringing the formerly separated devices together into a common partition. Another possibility is a mobile topic owner device which carries a data object from partition $\mathcal{A}$ to partition $\mathcal{B}$, comparable to a delay-tolerant network [61].

The trade-off for partition tolerance is the lack of guaranteed delivery: it is a-priori unknown if and when a data object can be transferred into another partition to the remaining device or if two partitions will merge. Therefore, no guarantees about delivery deadlines or delivery in general can be made. User-Centric Networking is only an option if intermittent inconsistency between devices is acceptable. The alternative would be to disable functionality altogether as soon as partitions are detected. These considerations are related to the CAP theorem [14] and are discussed in more detail in Chapter 6.

3.3.1 Example

Assume one topic t and two users A and B. User A is TO_t and allowed publisher, B is subscriber. Each user has two devices: $\mathbb{D}_A := \{a_1, a_2\}$ and $\mathbb{D}_B := \{b_1, b_2\}$.

Figure 3.4 gives a simple example for partition tolerance and how autonomous partitions keep their functionality. In the figure, there two different networks containing $\{a_1, b_1\}$ and $\{a_2, b_2\}$ respectively, i.e. one device of each user. Both networks are connected regularly, e.g. via the Internet (step ①). Here, a_1 publishes a new data object. Since the networks are connected, this data object can therefore be delivered from a_1 to the remaining three devices (step ②).

If the connections between the two networks breaks away (step ③), the two devices in the respective partition can still communicate with each other. a_1 now publishes another data object, which gets delivered to b_1, despite the missing connectivity to the other network (step ④).

As soon as connectivity returns between the two partitions, a_1 can deliver the new data object to the devices in the other network (step ⑤). Now, both data objects were successfully delivered to all four devices (step ⑥).

3.4 Resource Awareness

As discussed in Section 3.1.2, a published data object is delivered from the publishing device via TO device to subscriber device. If the device pool contains additional devices, there are multiple possibilities how the data object gets forwarded. Thus, a suitable *forwarding policy* must be defined (see Section 3.1.2).

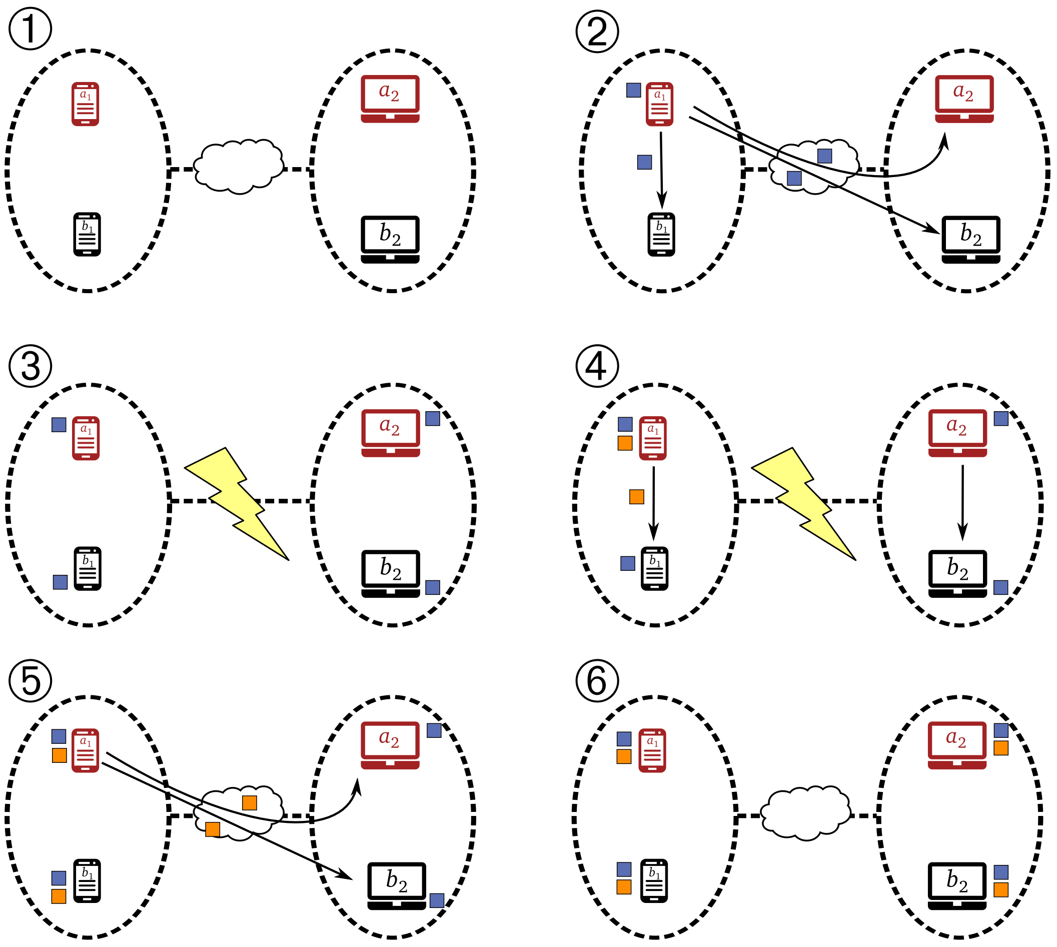


Figure 3.4: *Partition tolerance in User-Centric Networking*

A forwarding policy is **resource-aware** if it takes the devices' available resources into account and bases to the process of data object forwarding and delivery on them. Both device availability and device capability can be input parameters here.

3.4.1 Examples for resource-aware forwarding

The following example shall illustrate resource-aware forwarding. Figure 3.5 displays the following scenario: publisher P has one device p_1 and uses it to publish δ_t. There is one TO device t_1 and a total of three subscriber devices: u_1 and u_2 of subscriber U, and v_1 of subscriber V.

Assume that device u_2 has a higher capability than all the other devices. Figures 3.5(a) to 3.5(c) display how data delivery can be more efficient, if the TO device t_1 is aware of this higher capability and a different forwarding policy is chosen.

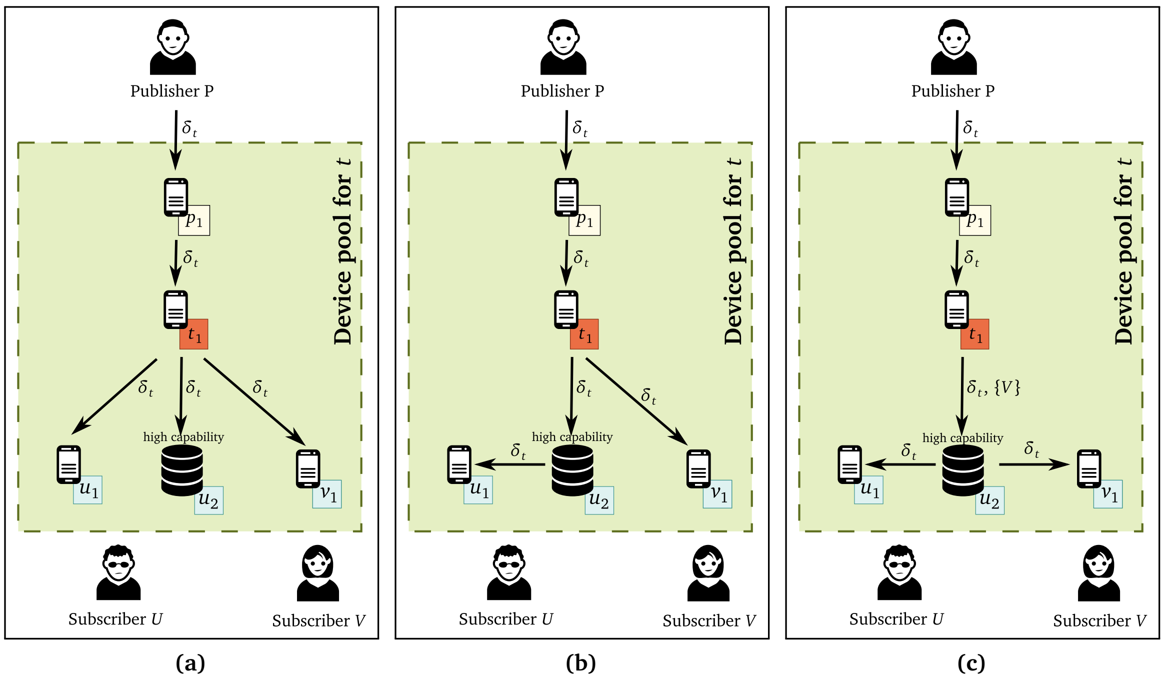


Figure 3.5: *Three examples for forwarding policies*

Forwarding policy (a)

In Figure 3.5, t_1 is not aware of u_2's higher capability. t_1 delivers δ_t to each device by itself, i.e. performs three deliveries. With this forwarding policy, t_1 has to perform as many deliveries as there are subscriber devices. Given that t_1 is a smartphone with low capability, this is an unnecessary effort for t_1, as will be discussed for Figures 3.5(b) and Figure 3.5(c).

Forwarding policy (b)

In Figure 3.5(b), t_1 is aware of u_2's higher capability. t_1 delivers δ_t to u_2 and u_2 delivers δ_t to u_1. Since only t_1 knows that V is also a subscriber, t_1 has to deliver δ_t to v_1 as well. Therefore, t_1 only performs two deliveries and one delivery is performed by u_2. With this forwarding policy, t_1 has to perform as many deliveries as there are subscribers.

Forwarding policy (c)

In Figure 3.5(c), t_1 is aware of u_2's higher capability. t_1 delivers δ_t to u_2 and while doing so, passes the information to u_2 that V is also a subscriber. With this information u_2 delivers δ_t not only to u_1, but also to v_1. Therefore, t_1 only performs one delivery and two deliveries are performed u_2. With this forwarding policy, t_1 has performed the minimum number of deliveries, while v_1 with the high capabilities has performed the rest. When regarding the overall resources of the device pool, this is the most efficient solution.

However, this efficiency comes with a trade-off: subscriber U learns that V is also a subscriber. Therefore, t_1 gives away application metadata which results in lesser privacy for V. This aspect is discussed in the next section.

3.5 Trust Dependencies

This section revisits the considerations from Section 2.3.3. There, the trust dependencies between group members were discussed, with third-party providers being involved. The trust dependencies were discussed with regard to four protection goals: content data confidentiality, content data integrity, content data availability and application metadata privacy. Some of these trust dependencies were between the group members, but also between group members and the third-party provider.

In User-Centric Networking, third-party providers are non-existent. Instead, the devices of group members become provider devices. This results in additional trust dependencies between the group members, which are discussed here.

3.5.1 Review from Chapter 2

First, the trust dependencies between group members are reviewed when third-party providers are involved.

As can be seen in Figure 2.8 on page 39, the only trust dependencies between group members are with regard to content data confidentiality and application metadata privacy:

- **Content data confidentiality**: It was argued that each member can access a data object which is shared within the closed group. Therefore each group member has to trust each other group member to keep this data object confidential.

- **Application metadata privacy**: Here, the group members also need to trust each other, since every group member can learn which other group member is involved in the communication process – with one exception: the subscriber does not have to trust the publisher, since the publisher does not know which users are subscribers. On the other hand, the publisher is known to the subscribers. Thus, the publisher needs to *transitively trust* all subscribers – the subscribers are unknown to the publisher and yet he has to trust them, therefore the publisher trusts that the TO has made a "good selection" of allowed subscribers.

For the other two protection goals – content data integrity and content data availability – no trust dependency between the group exists, since no group member device is a provider device. Therefore, no group member device delivers data objects to subscriber devices. Instead, these trust dependencies are between group members and third-party providers. This changes in User-Centric Networking – since no third-party providers exist here, these remaining two protection goals become dependencies between the group member as well.

3.5.2 Additional trust dependencies

Figure 3.6 shows an adaption for User-Centric Networking of Figure 2.8.

Here, the third-party provider was removed. Instead, the TO assumes the central role for data delivery, as was discussed in Section 3.1.2: the TO accepts a published data object from the publisher and delivers it to the subscriber. Therefore the following two additional trust dependencies arise:

- **Content data integrity**: If using cryptographical signatures or Message Authentication Codes (MAC), the publisher and subscribers in a UCN do not have to trust the TO, since weak integrity is secured. Otherwise, blind trust by the publisher and subscriber in the TO is required.
- **Content data availability**: Publisher and subscribers need to blindly trust the TO to deliver the data object.

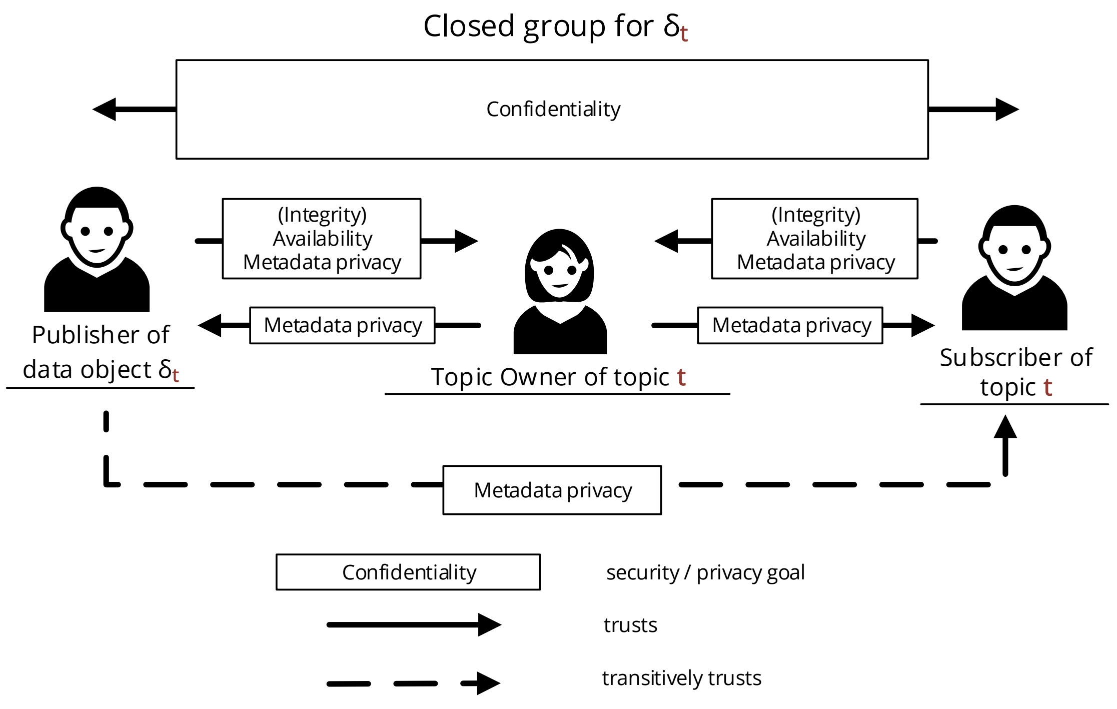


Figure 3.6: *Trust dependencies between group members in User-Centric Networking*

3.5.3 Helping subscribers

Figure 3.6 assumes that all data objects get delivered by the TO herself. In the context of resource-aware forwarding policies it was discussed, that once a published data object was approved by the TO, subscribers can help the TO to deliver the data object to other subscribers. See Section 3.4.1 and Figure 3.5(c) for an example.

In such a case, not only TO devices, but also subscriber devices become provider devices. Therefore, additional trust dependencies between subscribers exist. See Figure 3.7. Without loss of generality, subscriber 1 helps the TO in delivering data objects to subscriber 2. Therefore, the same trust dependencies from subscriber 2 to subscriber 1 exist as from subscriber 2 to the TO.

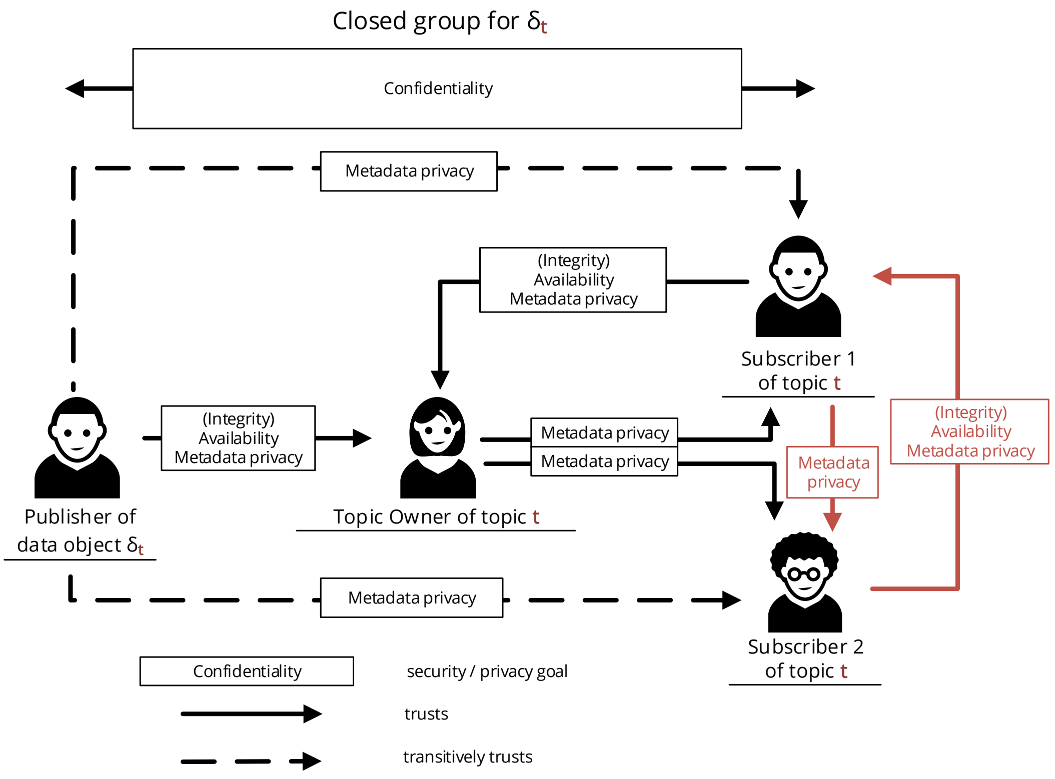


Figure 3.7: *Trust dependencies between group members in User-Centric Networking, with helping subscribers*

In Section 2.3.3, it was discussed that in U2U communication, receiver anonymity between two subscriber is generally possible. With helping subscribers, receiver anonymity between the two given subscribers is not possible anymore.

3.6 Privacy Model

This section covers the different motivations in exploiting information about users via application metadata. To this end, four different adversary types are presented – both for third partys as well as for group members. Based on these adversary types and different provider schemes, a privacy model is created. The model introduces an order from "low" to "high" privacy, by displaying which provider scheme allows for / excludes which types of adversary.

3.6.1 Adversary Types

There are different motivations for either a third-party provider or a group member to gain information about one specific or all members of a closed group. Let A be an *adversary* – a third-party provider or group member that infers and exploits PII of the (remaining) group members against their will or knowledge. Let T be the TO of a closed group. Let U be a member of this closed group (U and T may be identical). Let U be the target of an adversary A. A can be one of the following four types.

- *Large-scale data miner:* A is a CSP or a FSP with many customers. A is not interested in U specifically, but regards all customers. A exploits the customers' PII to learn more about them and their behaviour. Such a behaviour might be economically motivated (e.g. for presenting personalized advertisments or for adjusting the service to the customers) or for reporting generally suspicious behaviour to governmental authorities.
- *Personal attacker (unknown to T):* A is a third-party provider. A does not know T personally, but is interested in U and U's communication patterns specifically. This adversary type can appear for CSP, FSPs and DHT.
- *Personal attacker (contact of T):* A is a third-party provider that T specifically chose for trust reasons, given that the chosen provider scheme enables such a specific choice. A knows T personally. A is interested in U and U's communication patterns specifically. This adversary type can appear for different third-party provider schemes, such as FSPs (if the FSP is controlled by A), DHTs (if they have additional social mechanisms, e.g. that friend nodes are preferred) or schemes that rely on general (not per-data-object) trust towards a contact. Examples for the latter will be presented in Section 3.7.2.
- *Group member adversary:* This adversary A is a member of the same closed group as U. A and U both know T and may or may not know each other. A can be an allowed publisher or subscriber that wants to learn about the (other) subscribers in general. Alternatively, A is interested in U and U's communication patterns specifically. This adversary type can appear whenever group members can infer which other users are group members, for example by a self-sufficient provider scheme with helping subscribers (see Section 3.4.1) or by application.

3.6.2 Privacy Levels

Section 3.6.1 presented different adversary types for third-party providers, as well as one group member adversary type.

Based on this groundwork, four different **privacy levels** are presented in this section. The idea is here that a specific TO owns the topic for a specific closed group and selects a service for exchanging data objects for this topic. This service is bound to a specific provider scheme – either a third-party provider is involved or a self-sufficient provider scheme is chosen. Depending on the provider scheme, different adversary types are possible. Based on this dependency, four different privacy levels are derived.

The derivation of the privacy levels stems from the considerations that were made in Section 2.3 for third-party providers and in Section 3.5 for UCNs.

The higher the privacy level, the more a respective provider scheme relies on users that are actually known to the topic owner or that are even meant to receive a published data object. Assuming that such users are more trustworthy than unknown providers to respect the privacy of the closed group, an increase in the privacy level results in a better privacy for the closed group. Even if users known to the topic owner cannot be trusted with this regard, different adversary types can be identified for different privacy levels and certain adversary types can be neglected.

- **I - Unknown to TO:** The provider scheme involves a CSP or third-party providers (FSP, DHT) that do not know the TO personally and vice-versa. The possible adversary types on this level are large-scale data miners and personal attackers (unknown to TO). This privacy level is classified as lowest due to the involvement of generally unknown parties.
- **II - Contact of TO:** The provider scheme involves only decentralized third-party providers (FSP, DHT) which the TO knows and generally trusts about not being malicious. Due to the TO's knowledge and trust about the third-party providers, this privacy level is classified higher than Level I. However, note that other adversary types are now possible: personal attackers (contact of TO) and group member adversaries. The latter type is possible when the provider is at the same time a group member. These adversary types differ from the adversary types on Level I and the TO might have reason not to trust specific contacts with regard to these adversary types. In this case, the TO must either select privacy level I (and merely exchange adversary types) or aim for a higher privacy level of III or IV, which excludes adversary types altogether.
- **III - Closed group members:** The provider scheme involves no third-party providers, but is self-sufficient. Therefore, only group member adversaries are possible. This privacy level allows for subscribers that help the TO to deliver data objects to other subscriber devices (see Section 3.4.1). In such a case, subscribers can learn about other subscribers and therefore can exploit that information.
- **IV - Topic owner and single group member:** The provider scheme involves no third-party providers, but is self-sufficient. Additionally, the TO first accepts a data object from the publisher and then passes it to each subscriber. The TO is the only user who keeps knowledge about allowed publishers and (allowed) subscribers. No helping subscribers are involved, i.e. a forwarding policy as in Figure 3.5(c) is excluded. This way, no group member other than the TO can learn which other users are also subscribers. All adversary types as discussed in Section 3.6.1 are now excluded altogether.

Privacy levels I and II include third parties, while privacy levels III and IV exclude third parties. Therefore, privacy levels III and IV require self-sufficiency as the provider scheme. In turn, a self-sufficient provider scheme reaches at least privacy level III. It depends on the exact design of a self-sufficient DDP, whether privacy level IV can be achieved.

In the examples for forwarding polices (Section 3.4.1), forwarding policy (a) and (b) have privacy level IV, while forwarding policy (c) has privacy level III.

The difference between privacy level III and IV can also be illustrated by the following analogy: assume a secret person A wants to tell multiple persons $B, C, D \ldots$. There is a difference if a) A gathers $B, C, D \ldots$ in one room and tells the secret to all of them at the same time or b) A tells the secret first to B, then to C, then to D, $\ldots$ one after another with the others being absent. In case a), all recipients learn which other persons are recipients. In case b), each recipient is unaware of any other recipients. In this analogy, person A is the the TO and $B, C, D, \ldots$ are subscribers. Case a) matches to privacy level III, case b) matches to privacy level IV.

Figure 3.8 presents an overview. On the left, the four privacy levels are presented as concentric circles, depicting the reduced set of possible storage devices. On the right, it shows how reducing the possible storage devices also increases the level of privacy / decreases the types of possible adversaries.

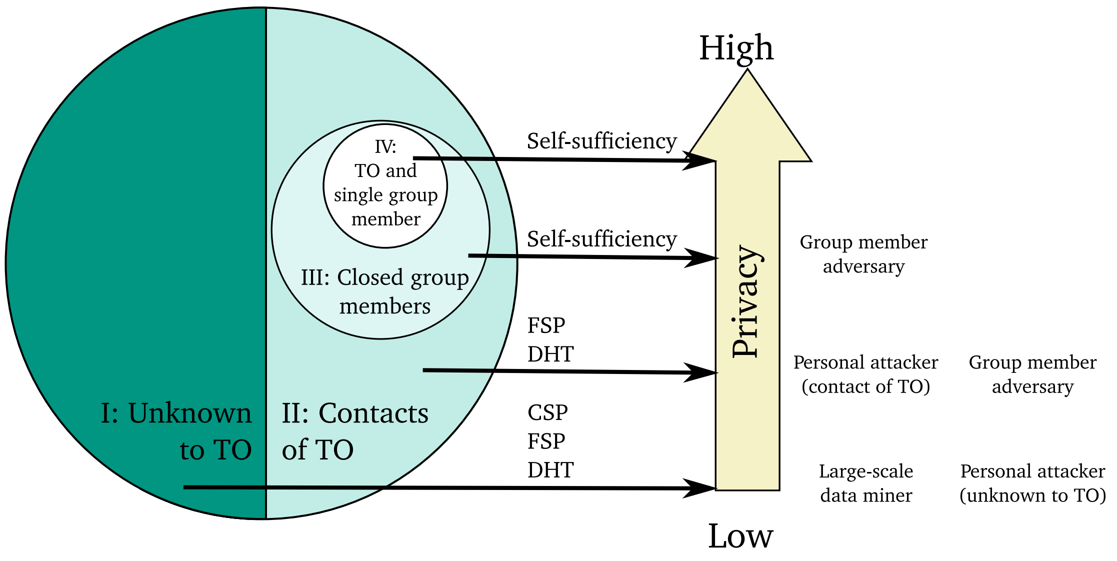


Figure 3.8: *Privacy levels*

3.7 Related Work

This section presents three types of related work. First, other approaches to metadata privacy besides self-sufficiency are presented. Second, U2U communication approaches where data is stored on generally trusted devices are presented. These involve third-party providers, but they are more similar to User-Centric Networking than the approaches that were presented in the last chapter (see Section 2.5.2). Third, different usages of the term "User-Centric Networking" are highlighted.

3.7.1 Other approaches to metadata privacy

Throughout Chapter 2, there was the central assumption that a) each user has an identifier which is used for topic owners to set access rights and b) if third-party providers are involved, this identifier is used as explicit application metadata to enforce access rights and to deliver data objects to the correct subscribers. Since application metadata that gets exposed to third-party providers poses a privacy risk to the users, there are different considerations how metadata can be hidden from third-party providers.

One possibility is *third-party reduction*: here, a data object δ_t is passed through less third parties on its way from publisher to subscriber, i.e. less third parties get in touch with the metadata. Self-sufficiency as a provider scheme achieves exactly this - it not only reduces, but excludes third parties altogether.

This section describes two **privacy enhancing techniques (PETs)** that aim *third-party anonymity* [22]: metadata is not revealed to third parties, while publisher and subscribers know with high certainty each other's identities. Thus, third parties are not excluded, but publisher and subscribers identifiers are hidden from them. The two presented approaches towards third-party anonymity are *mix networks* and *broadcast schemes*.

Mix networks

With the help of mix networks, each message gets forwarded through multiple third-party relays, which obfuscates the connection between publisher and subscriber from the eyes of a single third-party provider, i.e. achieves unlinkability.

Figure 3.9a displays *onion routing* as one popular use-case for mix networks. Here, A is the publisher and B the only subscriber. A encrypts the data object δ with B's public key k_B, so that only B can read δ. The results is $k_B(\delta)$. Additionally, A selects a route via C's device c_1 and D's device d_1. Going backwards from the destination (B's device b_1), A tells each hop the next device hop and encrypts this information with the hop's public key.

This way, C only learns that it shall forward the data object to d_1, but does not know the actual destination, since it cannot decrypt the information that was encrypted with k_D. Similarly, D only learns that it shall forward the data object to b_1, but does not learn that the data object's origin is A. Neither C nor D can read δ since it is encrypted with k_B.

Disadvantages of mix networks is the additional required infrastructure and effort. Additionally, effective unlinkability requires a high number of participants ("*anonymity loves company*" [24]).

Note that a mix network can also help to achieve sender anonymity and receiver anonymity. Sender anonymity is out of scope for U2U communication, since each data object is linkable to its publisher. Receiver anonymity can also be achieved with self-sufficiency.

Broadcast schemes

Figure 3.9(b) shows a form of third-party anonymity via a DDP which uses a broadcast scheme. Let D be a third-party provider that is responsible for delivering data objects. Here, A does not have to disclose the subscriber B to D for successful delivery of δ, since d_1 simply broadcasts δ to all known devices. Given that δ is encrypted with B's public key k_B, only b_1 and b_2 can successfully decrypt δ. All other receiving non-subscriber devices (e.g. $\mathbb{D}_C$) fail at deciphering δ and discard δ. Since D does not know which devices succeed at decryption, D cannot learn that A is communicating with B. This approach achieves at least receiver anonymity.

Disadvantages of broadcast schemes are lack of scalability, specifically since it profits from a high number of participants, similar to mix networks. For example, if A and B have a private message communication and their devices alternate in sending data objects to d_1, D can assume a communication process merely based on the timing how data objects arrive.

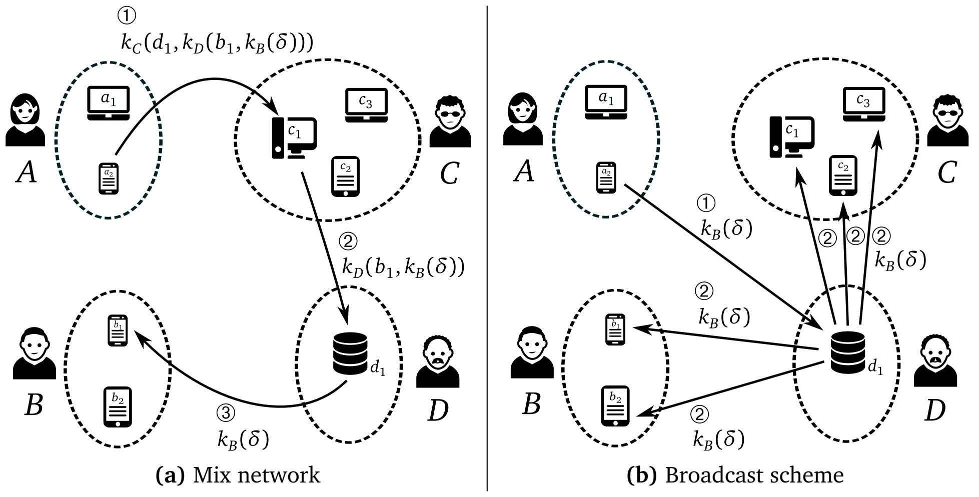


Figure 3.9: *Reducing metadata visibility by mix networks (left) and broadcast schemes (right)*

3.7.2 User-to-user communication via trusted devices

Besides the approaches for user-to-user-communication that were discussed in Section 2.5.2, there are other approaches where all data for a specific is stored on devices that the respective user generally trusts. These approaches do not fit into the CSP / FSP / DHT schema from Chapter 2. Instead they are more similar to User-Centric Networking and are thus discussed here.

- *Haggle* [107] is a generic framework which decouples mobile applications from the underlying network infrastructure. To this end, Haggle offers so-called *name graphs*, which maps a user's identity to multiple protocols and connectivity methods, such as an email address, phone number or Bluetooth MAC address. Application-specific data objects from applications are converted to generic data containers which can be transported via any connectivity opportunity that arises first. Thus for example, one user can send an email directly from his personal device to another while having ad-hoc network connectivity only. While this offers an elegant form of partition tolerance, Haggle lacks the resource-awareness and privacy-awareness of User-Centric Networking. According to the authors, "*security and privacy have not been addressed as key concerns*". Instead, Haggle's focuses on the usage of heterogenous network interfaces – if a third-party provider can be leveraged, it is used. Thus, only privacy level I is reached.

- Mega et al. [72] propose a decentralized online social network (DOSN) where each device only communicates with devices of contacts and contact's contacts. This approach is more relaxed than User-Centric Networking: based on the classic idea of a OSN, all data objects bound to a given user shall be delivered to all his contacts. There is no notion of per-user-object closed groups. While additional encryption-based access control schemes could be added on top, each user's contacts can communicate directly with other contacts for exchanging the encrypted data objects. Thus only privacy level II is reached. Furthermore, multiple devices per user are not regarded by the authors.

- *Confidant* [68] is a DOSN where each user stores his data on his own personal devices such as PCs, so-called *storage servers*. For better availability, his data is replicated on the storage servers of some of his contacts, so-called *replicas*. Similar to Mega et. al [72], *Confidant* does not have the notion of closed groups on a per-data-object basis: "*Each Confidant user runs a storage server that contains plaintext copies of all of her objects and access policies. [. . .] A storage server may also act as a replica for another user if the other user trusts the server's owner to 1) read all of her [unencrypted] objects [and] 2) enforce access policies [. . .]*" Thus, replicas are hand-picked by the user from the contacts the user generally trusts most. If a user would want to split his data, e.g. to distinguish between work data and personal data, he would have to run two separate storage servers, each with his own replica set. These restrictions do not apply to User-Centric Networking where self-sufficiency on a per-data-object basis is achieved. Confidant reaches only privacy level II.

- *SuperNova* [99] is another DOSN where a user's data is preferably stored on the devices of trusted contacts. Furthermore, SuperNova aims at handling device heterogeneity by relieving from device strain and instead offloading data objects to unknown third-party super-peers. Thus, SuperNova reaches only privacy level I.

- *Vis-à-Vis* [97] is a location-based DOSN where each user operates a highly-available cloud server under the user's control, which stores only the user's own personal data (*Virtual Individual Server, VIS*). Vis-à-Vis allows to search for other users in the same region. To this end, VIS of the same region are grouped hierarchically in tree structures from city blocks to countries. Search queries over a region are sent down the respective tree to the VIS. The main disadvantage of this approach are the operational costs for each user's mandatory VIS. Unlike in User-Centric Networking, users' personal devices cannot be leveraged and there is no partition tolerance.

- *Safebook* [21] is a DOSN where a user's data is only stored on his own device and those of his trusted contacts. Assuming that the user in the center of a matryoshka (see Section 2.5.2) is the topic owner, all his contact's devices act as provider devices. Thus, privacy level II is reached. Note that Safebook also leverages a DHT to allow non-contacts reach matryoshka entrypoints, in order to access public data. However, this feature is out of scope for U2U communication.

3.7.3 Other definitions of User-Centric Networking

The term *User-Centric Networking* is occupied in multiple ways, partly with vague definitions. This section highlights other definitions and contexts of this term and how these differ and concur with the usage in this thesis.

[105] defines UCNs and the central role of the users in them as follows. The authors specifically mention privacy in form of trust and resources in form of connectivity as issues which are inherently tied to UCN:

> *User-centric networks (UCNs) are a recent architectural trend of self-organizing autonomic networks where the Internet end user cooperates by sharing network services and resources. UCNs are spontaneous and grassroots deployments of wireless architectures, ad hoc or infrastructured, often involving low-cost equipment. [. . .]*
>
> *The new role of an empowered end user is disruptive in several aspects:*
>
> - *In the end-to-end Internet paradigm, an end-user device will actively participate as a network element in addition to being an endpoint host.*
> - *In network boundaries of trust, these will need to be extended in a way that should mimic social behavior.*
> - *In service continuity, end-user devices devices should be capable of handling intermittent Internet connectivity as well as fast and transparent roaming between micro-operators.*

The authors emphasizes spontaneous wireless communication and infrastructure service provision, such as providing a shared internet access to other people in the neighborhood (also called *user-provided networking*). Other scenarios are network performance measurements and software-defined networking. These scenarios are vastly different from the sharing of data objects: while user-provided networking regards OSI layers 2 and 3, this thesis focuses on layer 7 applications that are controlled via a user interface. Furthermore, this thesis assumes that data exchange from one device to another can be realized by connectivity of all kinds, as long as two device are available for each other: wired, wireless, infrastructured, ad-hoc, via the Internet or by local networking only. Privacy and lack of device availability are central considerations of this thesis, which are also taken into account by [105].

[117] defines a User-Centric Network differently and very closely to delay-tolerant networks: "an intermediately connected mobile social network, a typical delay-tolerant network that employs human beings' social characteristics for information dissemination"

The notion of user-provided networking, and its implications such as trust and incentives, is also the basis for a Dagstuhl seminar on User-Centric Networking [89]. However, further areas where user's personal devices can be leveraged were explored, such as analyzing user behaviour and mobility models.

One discussed aspect of the seminar – further elaborated in [20] – was today's Internet reality with its stark separation of centralized hosted services and smart personal devices simply as "the Cloud". The authors compare this situation to the 1970s where dumb terminals were used to merely display data that was stored and processed on mainframes. Since personal devices are vastly more powerful nowadays, the authors deem the current situation as an unnecessary waste of resources at the users' fingertips. They emphasize the active research in *"ultra-distributed systems, such as peer-to-peer file sharing, swarms, ad-hoc mesh networks, mobile decentralized social networks*" as an alternative to the Cloud, summarizing such systems as "the Mist". The authors assume privacy by default since the users hold their data on their own devices only and have full control on what they share. However, this assumption is a strong restriction on how data is stored and replicated in the Mist and may have a negative impact on data availability.

The authors' vision of "the Mist" is arguably closest to the definition of User-Centric Networking in this thesis: leverage of personal devices in order to circumvent the need for third parties. However, the term of "User-Centric Networking" is significantly more specific – including the three design goals of self-sufficiency, partition tolerance and resource-awareness, as well as a formal definition for a UCN.

3.8 Conclusion

This chapter presented User-Centric Networking for U2U communication as an alternative to third-party provider schemes that were earlier presented in Chapter 2. User-Centric Networking is based on self-sufficiency, which is the idea that delivering data objects from the publisher to the subscriber shall be handled by the personal devices of the closed group members only. The main advantage of this approach is an improvement of the closed group members' privacy, since application metadata is not visible to a third party. The main disadvantage is that all efforts for data object delivery cannot be outsourced to a third party, but need to be handled by the personal devices of the closed group members.

Section 3.1 introduced to User-Centric Networking and self-sufficiency. It presented a formal definition for a User-Centric Network (UCN) and discussed the central role of the topic owner per closed group.

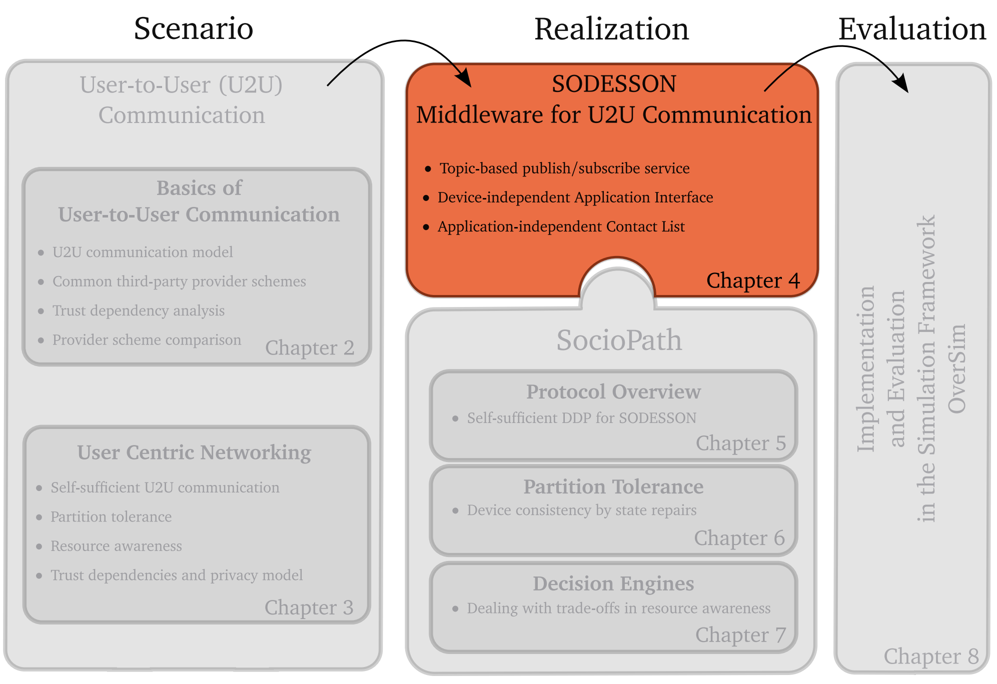
Personal devices are heterogenous with regard to device availability and device capability. To illustrate this, three example device classes that differ in this regard were presented (Section 3.2). User-Centric Networking follows two design goals that take such heterogeneity into account: first, partition tolerance shall handle intermittent lack of availability between devices and let them recover from missed data object deliveries (Section 3.3). Second, resource awareness shall take devices' availabilities and devices' capabilities into account for the forwarding policy (Section 3.4).

In Section 3.5, trust dependencies between the closed group members in User-Centric Networking were analyzed, similar to the analysis in Chapter 2 for third-party provider schemes. Based on these two analyses, a privacy model with four distinct privacy levels was derived (Section 3.6).

Finally, Section 3.7 discussed related work for privacy-aware U2U communication with approaches similar to User-Centric Networking. However, none of these approaches follow the notion of closed groups on a per-data-object basis. Instead, they rely on a general trust in friends and friends' devices, regardless whether these friends are actually subscribers for a given data object.

Chapter 4

SODESSON Middleware



Scenario
Realization
Evaluation
User-to-User (U2U) Communication
Basics of User-to-User Communication
U2U communication model
Common third-party provider schemes
Trust dependency analysis
Provider scheme comparison
Chapter 2
User Centric Networking
Self-sufficient U2U communication
Partition tolerance
Resource awareness
Trust dependencies and privacy model
Chapter 3
SODESSON
Middleware for U2U Communication
Topic-based publish/subscribe service
Device-independent Application Interface
Application-independent Contact List
Chapter 4
SocioPath
Protocol Overview
Self-sufficient DDP for SODESSON
Chapter 5
Partition Tolerance
Device consistency by state repairs
Chapter 6
Decision Engines
Dealing with trade-offs in resource awareness
Chapter 7
Implementation and Evaluation in the Simulation Framework OverSim
Chapter 8

Figure 4.1: *Placement of Chapter 4 in the big picture*

4.1 Introduction

This chapter describes SODESSON – a middleware which provides a generic service to U2U communication. This service is the first building block for a User-Centric Networking implementation, as presented in Chapter 1. Figure 4.1 places the SODESSON middleware into the overall concept of User-Centric Networking.

As a middleware, SODESSON runs on each device which is meant to participate in U2U communication, i.e. where the device used for running U2U applications and be an addition to the respective closed groups' device pools. It is positioned as a middle layer between U2U applications and the OS network stack.

SODESSON provides a generic **topic-based publish/subscribe service** for all U2U applications. SODESSON fulfills the following three requirements:

- **Application interface:** SODESSON's service features an application interface which understands and handles U2U entities as defined in Chapter 2.1. The purpose of the interface is two-fold: first, it accepts data objects published by applications for a given *topic*. Multiple applications may run at the same time on one device, which requires a multiplexing mechanism via the interface. Second, the interface is used for defining access rights: each topic is bound to a specific user, the *topic owner*. Via the application interface, he defines the set of allowed publishers and allowed subscribers per topic he owns. The interface also allows users to subscribe to a given topic.
- **Abstraction from devices:** In *user-to-user* communication, an application should not have to care about addressing the "correct" device (i.e. the one which a user controls to generate and consume application data). This is especially the case in User-Centric Networking, where it is assumed that each user can have multiple devices. The level of abstraction on which SODESSON and thus the application interface operate, is topics and user: data objects get published for a specific topic and topics get subscribed. A topic is uniquely identified by the combination of an arbitrary string and a topic owner, i.e. a user. On this level of abstraction, no devices are involved.
- **Contact management:** SODESSON features an application- and device-independent *Contact List* (see Section 4.4.1) for each user. This contact list is replicated on each of the respective user's devices. A contact list holds entries of other users that mutually agreed with the given user to be entries of each other's contact list. A topic owner selects the allowed publishers and allowed subscribers for a topic as subsets from his contact list.

With these components, SODESSON merely provides a generic, device-independent U2U communication service for applications running on a single device. The middleware is complemented by a **Data Distribution Protocol (DDP)** which handles inter-device communication and is responsible for delivering data objects from a publisher's device to the subscriber devices. For SODESSON, this DDP is exchangable and can be freely defined – any provider scheme is possible, even a centralized approach would be conceivable.

In User-Centric Networking, data delivery is self-sufficient, therefore SODESSON needs to be complemented with a self-sufficient DDP. As Figure 4.1 indicates, this requirement is addressed by SocioPath – a self-sufficient DDP for SODESSON, which is to be presented in the Chapters 5 to 7.

4.2 Basic Concept and Architecture

SODESSON enables a U2U application on device a to publish a data object δ_t with topic t. The application passes δ_t together with t to the middleware via the application interface. Here, δ_t is

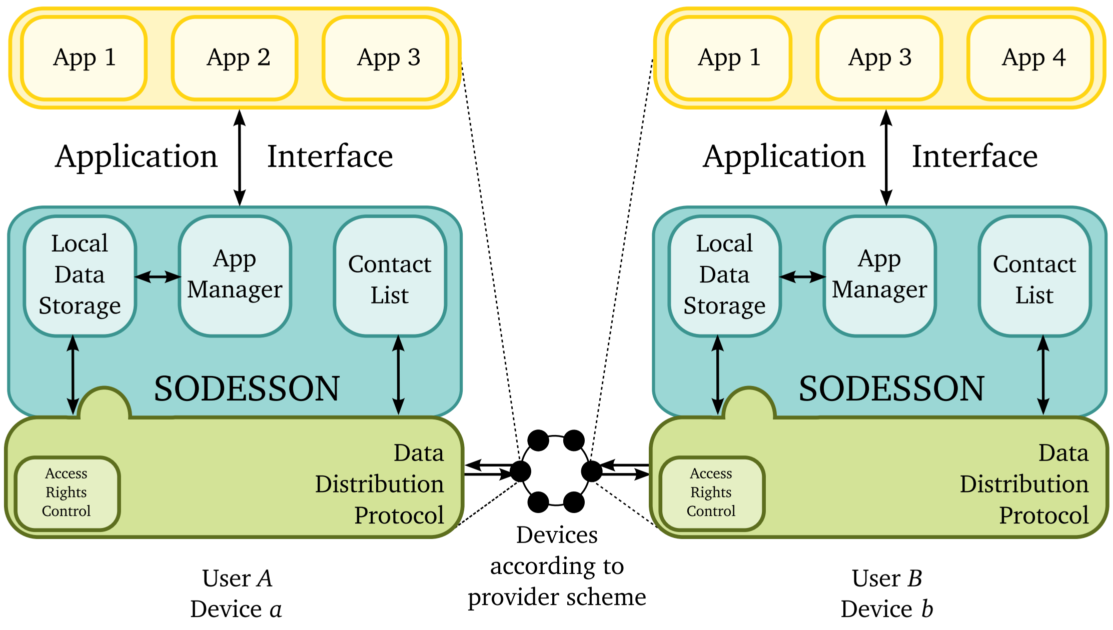


Figure 4.2: *SODESSON overview*

either passed directly as a binary value or indirectly as a reference to a file on the local filesystem. Together with the DDP, SODESSON makes a best effort to deliver δ_t from a to $\mathbb{D}_{\mathbb{S}_t}$ with $\mathbb{S}_t$ being the subscribers of t and $\mathbb{D}_{\mathbb{S}_t}$ being the sum of their devices (see Table 3.2 for a list of the used symbols).

Figure 4.2 depicts two devices a and b, each running a different set of applications and the SODESSON middleware. a runs App[1] 1, App 2 and App 3, while device b runs App 1, App 3, App 4. Still, all applications communicate with SODESSON via the same *application interface*. It is used to publish and receive data objects via SODESSON. Published and received data objects on a device are persisted by the **Local Data Storage**: here, a data object is either stored directly in this data structure or a reference to a file on the local file system is made.

The **App Manager** handles the local mapping between application and subscribed topics: If a user subscribes to a specific topic t with an application a on a device d, the App Manager running on d stores the connection betweeen a and t. As soon as a published data object δ_t with topic t arrives at d, the App Manager passes δ_t to the correct application(s). Multiple apps can be mapped to one topic and vice-versa. Thus, the App Manager is a multiplexer between application(s) and topic(s).

4.2.1 Tasks of the DDP

The *Data Distribution Protocol (DDP)* is responsible for two tasks: first, enforcing the access rights, i.e. making sure that only data objects that were published by allowed publishers get delivered to the subscribers (and only these). Second, delivering the data object itself. Both tasks are to be

[1]Both the terms "app" and "application" are used in this thesis. "Application" is mostly used in generic contexts, while an "app" is an application from the point of view of the SODESSON middleware.

performed by specific provider devices, depending on the DDP's provider scheme. In SODESSON's design, it is assumed that each of the two tasks can be performed by different devices.

4.3 Publish/Subscribe Service

This section covers the topic-based publish/subscribe service SODESSON provides to applications. It is divided into the following subsections:

Section 4.3.1 describes how the entities in U2U communication are represented in SODESSON.

In Section 4.3.3, the publish/subscribe workflow is described in detail with the help of a concrete example (blog post as a data object). It is described how the different components of SODESSON are used in order to deliver a data object from the publisher device to a subscriber device.

Afterwards, in Sections 4.3.4 and 4.3.5, the structure of the two modules App Manager and the Local Data Storage are described. With the help of these modules, additional publish/subscribe specifics are described: Retrieving (Section 4.3.6), updating and deleting (Section 4.3.7) a published data object.

See Appendix A for a full reference on the publish/subscribe methods of the application interface.

4.3.1 Representing U2U Entities in SODESSON

Section 2.1 presented the main U2U entities users, topics and data objects in an abstract manner. This section describes how these entities are uniquely identified and addressed in SODESSON. A summary overview is shown in Table 4.1.

- **User:** Each user A generates his own public/private keypair ($pubkey_A$/$privkey_A$) for signed and encrypted communication. A is globally uniquely identified by other users via $pubkey_A$. In addition, a shorter user ID $UID_A := hash(pubkey_A)$ can be generated by all users who know $pubkey_A$ via a system-wide fixed hash function. The hash function is assumed to be adequately collision resistant, hence UID_A is assumed to be as unique as $pubkey_A$. UID_A is a more space-efficient method to identify A.

  If A must generate a new keypair (e.g. because the old $privkey_A$ was compromised), his unique identification changes and needs to be updated in all relevant places.

- **Topic:** Each topic t consists of two parts, which are mandatory for addressing t: An arbitrary title string $title_t$ and the topic owner's $UID =: TO_t$. The title can be freely defined (e.g. by an application), whereas the TO_t part is used to identify the topic owner. Therefore, two topic owners can each own a unique topic with the same title. Each topic is assumed to be globally unique. Due to TO_t being a UID, this is an extension of the assumption *pubkeys* and *UIDs* being globally unique.

- **Data object:** Each data object δ_t is uniquely identified by a 3-tuple:
  - Topic t

– Object ID OID_{δ_t}

– Creation / update timestamp $time_{\delta_t}$

OID_{δ_t} is an integer value. It can be set by the application to a specific value in order to give δ_t a specific, application-internal, semantic meaning. If a specific value is not given by the application, SODESSON generates a random value. Here, the integer size is assumed to be large enough to avoid collisions (e.g. 64 bit)[2].

$time_{\delta_t}$ is initially the timestamp (e.g. Unixtime) of the creation of δ_t. If δ_t gets updated, $time_{\delta_t}$ gets updated with a current timestamp. The timestamp is determined by the current system time. All devices are assumed to have synchronous clocks which are regularly refreshed, e.g. via NTP [75].

Table 4.1: *U2U entities in SODESSON*

U2U Entity	Identifier	Explanation
User A	$\{pubkey_A, [UID_A]\}$	
	$pubkey_A$	A's unique public key.
	$UID_A := hash(pubkey_A)$	A's unique UID. Shorter alternative to $pubkey_A$ where space efficiency is required.
Topic t	$\{title_t, TO_t\}$	
	$title_t$	Topic title. Arbitrary string
	TO_t	Topic owner's UID, i.e. if A is topic owner $\Rightarrow TO_t = UID_A$
Data object δ_t	$\{t, OID_{\delta_t}, time_{\delta_t}\}$	
	t	Topic (as defined above)
	OID_{δ_t}	Unique object ID of δ_t (integer)
	$time_{\delta_t}$	Timestamp of creation or last update of δ_t.

4.3.2 Users vs. Applications

Note that a publisher (i.e. a human user) uses an application on one of his devices to publish a data object. Technically, the data object gets created and afterwards published by an application by demand of the user. Therefore, there is a congruency between the user and application. For the sake of simplicity, an instance of the respective application which is used by a publisher is called a *publisher* as well. The same applies to a subscriber (user) who uses an application to consume data objects in regard to the subscribed context. Here, the respective instance of the application is called a *subscriber* as well. For the terms *publishers* and *subscribers*, the distinction between user and application will be emphasized where required.

[2]Estimating the collision probability here is equivalent to the birthday problem [92]. To put the collision problem into scope for the 64 bit example: In a number space of 2^{64}, the probability, that any two in $1.9 \cdot 10^8$ data objects *with the same topic* have the same OID, is 0.1%.

4.3.3 Publishing a Data Object: Step by Step

This section covers a typical U2U application workflow. It describes how the users interact with their applications and how an application communicates with SODESSON and vice-versa via the application interface. The workflow consists of four steps:

(1) Binding an application to SODESSON

(2) Creating a new topic

(3) Subscribing to a topic

(4) Publishing a new data object and delivering it to the subscribers

By the end of these steps, one user (publisher and topic owner) has shared a data object with another user (subscriber) by using the SODESSON middleware. The steps present the most important methods of the application interface inside a closed group of users.

4.3.3.1 Step one: Binding an application to SODESSON

First, an application for U2U communication running on a device needs to be bound to the SODESSON middleware running on the same device.

The application calls the `registerApp` method of SODESSON's application interface (no arguments are passed). SODESSON creates a *handle*, a device-wide unique numeric identifier, for the application and returns it to the application.

The application uses the handle to identify itself in subsequent application interface method calls. The mapping of the handle to the application is stored by SODESSON in the module App Manager.

Example workflow

Assume that User A on device a has installed the application "*App 1*", which is an application for creating and reading private blogs. These private blogs have closed groups as an audience, i.e. the application is an example for U2U communication.

After "App 1" has called the `registerApp` method of SODESSON's application interface, SODESSON returns the numeric value 123 as the handle and stores the mapping "App 1" and 123 in the App Manager.

4.3.3.2 Step two: Creating a new topic

Prior to the next steps (i.e. subscribing to a topic or publishing a data object with a specific topic), a topic needs to be created in the first place. The topic's creator is always the topic owner. First, the topic requires a title. Additionally, the topic owner must define the sets of allowed publishers $\mathbb{P}_t$ and allowed subscribers $\mathbb{A}_t$ for the given topic t.

Optionally, the topic owner can define a privacy level (see Section 3.6.2) for a topic. This argument only has an effect if the DDP supports the demanded privacy level and can adapt to different demands. Note that the topic owner defines the privacy level for his topic, any other publisher or subscriber cannot influence this decision. The application calls updateTopic with the arguments as given in Table 4.2.

Table 4.2: *Arguments for updateTopic*

Data type	Argument	Description
`String`	title	Topic title $title_t$
`Array<UID>`	allowedPubs	UID list of allowed publishers $\mathbb{P}$
`Array<UID>`	allowedSubs	UID list of allowed subscribers $\mathbb{A}$
`Enum`	privacyLevel	*(Optional)* Demanded privacy level (1-4) as defined in Section 3.6.2.

As can be seen in Table 4.2, allowed publishers and allowed subscribers are passed as arrays of UIDs. Thus, before an application can call updateTopic, it requires the respective UIDs first. Since the sets $\mathbb{P}_t$ and $\mathbb{A}_t$ are subsets of the topic owner's contacts, the application needs to get the *Contact List*. Similar to the publish/subscribe interface, SODESSON provides methods for contact management to applications (see Section 4.4). One of these methods is getContacts, which returns the full Contact List, including UIDs. Therefore, before the application calls updateTopic, it calls getContacts first. Afterwards, the application must make its selection of the allowed publishers and allowed subscribers subsets or implement a possibility for the user to make this selection, e.g. via a graphical user interface (GUI). Finally, the application calls updateTopic and passes the selection as described above.

Next, the topic creation including the access rights needs to be indicated at a provider device, as these are responsible for enforcing the access rights of this topic. This inter-device communication is the task of the DDP. The addressed provider device updates its Access Control component accordingly.

Example workflow

Figure 4.3 depicts the workflow of this step using the private blog example. Assume that all blog posts will be published with the same topic. The application arbitrarily selects a topic title, in this case it is "*App1_Blog*".

User *A* wants to define the access rights for his private blog: he should be the only user who may publish blog posts, while his contact *B* should be able to read the blog posts. App 1 accesses the Contact List via the method getContacts ① and shows the list of contacts to *A* and lets him set the rights accordingly via a GUI.

A defines himself as an allowed publisher, *A*, *B* and *C* become allowed subscribers. For the privacy level, *A* selects 3. In order to fulfill this demand, a self-sufficient DDP is required (see Section 3.6.2), otherwise this argument is ignored by SODESSON. Since the privacy level is not 4, subscribers may learn about other subscribers. This way, *B* and *C* can help each other to distribute new blog posts, which *A* deems more relevant than a higher group members' privacy.

Therefore, App 1 calls the application interface method updateTopic with the arguments as shown in Table 4.3 ②.

Table 4.3: *Arguments for updateTopic (Blog example)*

Data type	Argument	Value in this example
`String`	title	"App1_Blog"
`Array<UID>`	allowedPubs	$\{UID_A\}$
`Array<UID>`	allowedSubs	$\{UID_A, UID_B\}$
`Enum`	privacyLevel	3

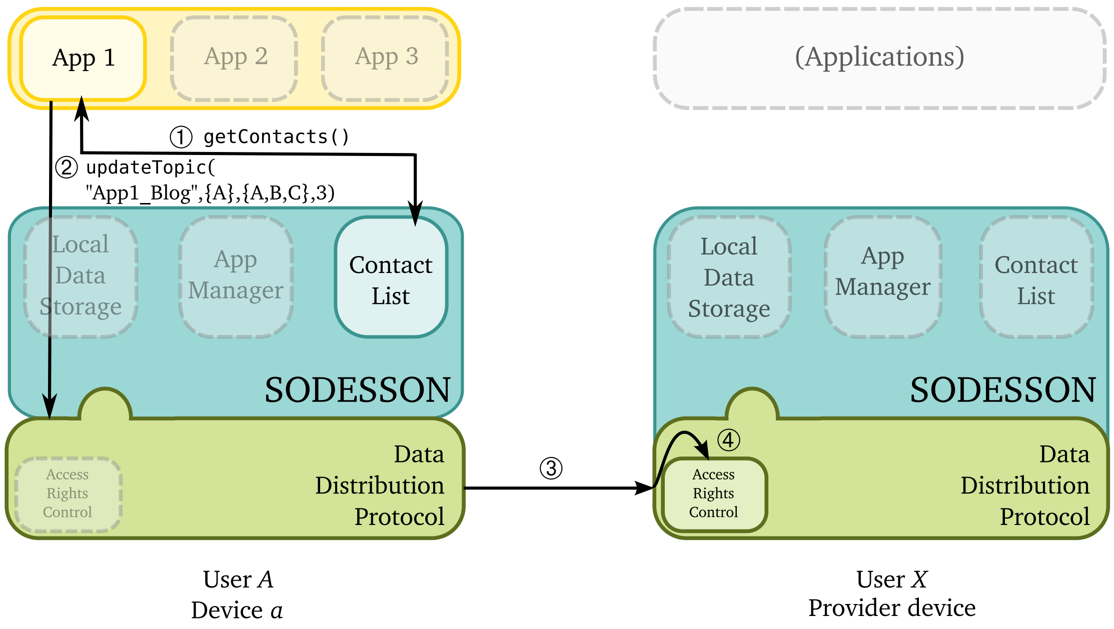


Figure 4.3: *Publish/subscribe, step two: creating a topic and defining the closed group*

Next, device *a* indicates the topic creation including the access rights at a provider device ③. The user who controls the provider device depends on the DDP's provider scheme – here the user is X. The provider device updates its DDP's Access Control component accordingly ④.

4.3.3.3 Step three: Subscribing to a topic

An application which requests to subscribe to a given topic $t = \{title_t, TO_t\}$ (see Section 4.3.1) must pass to SODESSON both $title_t$ and TO_t, as well as its handle.

The *handle* is the device-wide unique numeric identifier the application received from SODESSON by the registerApp call (see step one). The handle is required for the App Manager to identify the subscribing application and store the respective mapping of application and subscribed topic. This way, multiple applications can subscribe to the same topic.

Therefore, the application calls the method subscribe with the arguments as given in Table 4.4.

The App Manager contains a data structure used for mapping applications to subscribed topics (see Section 4.3.4) via the application's handle. With this information, data objects can be delivered to the correct applications. After a call of the subscribe method, the App Manager *prematurely*

Table 4.4: *Arguments for* subscribe

Data type	Argument	Description
`String`	topicTitle	Topic title $title_t$
`UID`	topicOwner	UID of the topic owner TO_t
`Integer`	handle	Application's handle

(i.e. without the subscription being confirmed by a provider device yet) stores the application via the passed handle and the requested, to-be subscribed topic in its data structure. This is done, so that a subscription success response can be later passed to the respective application.

Next, the subscribe request needs to be indicated at a provider device, as these are responsible for enforcing the access rights of this topic. Therefore, SODESSON passes the subscription request to the DDP, which in turn forwards the subscription request to the responsible provider device(s). Each provider device checks if the requesting user is an allowed subscriber. If this is the case, the provider device updates its Access Control component and enters the requesting user as a subscriber.

The provider device now returns a response to the requesting device whether the subscribe request was a success or a failure. A successful response is handled by the App Manager which forwards it to all applications that have subscribed to the topic. If no successful response arrives at the application, it can retry to subscribe anytime.

Note: SODESSON does not provide a mechanism to notify users about being allowed subscribers. This needs to be handled by the application itself. For example, the application could define a dedicated topic for indicating permissions and the topic owner can publish a data object here to inform users about their publishing / subscription rights. As a simpler alternative, each user could simply try to subscribe without having any prior knowledge if he is an allowed subscriber at all.

Example workflow

Figure 4.4 depicts the workflow of this step by continuing the private blog example. Assume that user *B* has also installed "App 1" on his device *b* and has bound "App 1" to SODESSON, analogous to user *A* on device *a* in step one. The handle returned by SODESSON is 456. In step two, user *A* defined user *B* as an allowed subscriber for topic {App1_Blog, *A*}. Now *B* decides that he wants to read *A*'s blog and thus subscribe to said topic.

For subscribing, *B* uses App 1 on his device *b*, which in turn calls the application interface method `subscribe` ① with the arguments as shown in Table 4.5.

Table 4.5: *Arguments for* subscribe *(blog example)*

Data type	Argument	Value in this example
`String`	topicTitle	"App1_Blog"
`UID`	topicOwner	UID_A
`Integer`	handle	456

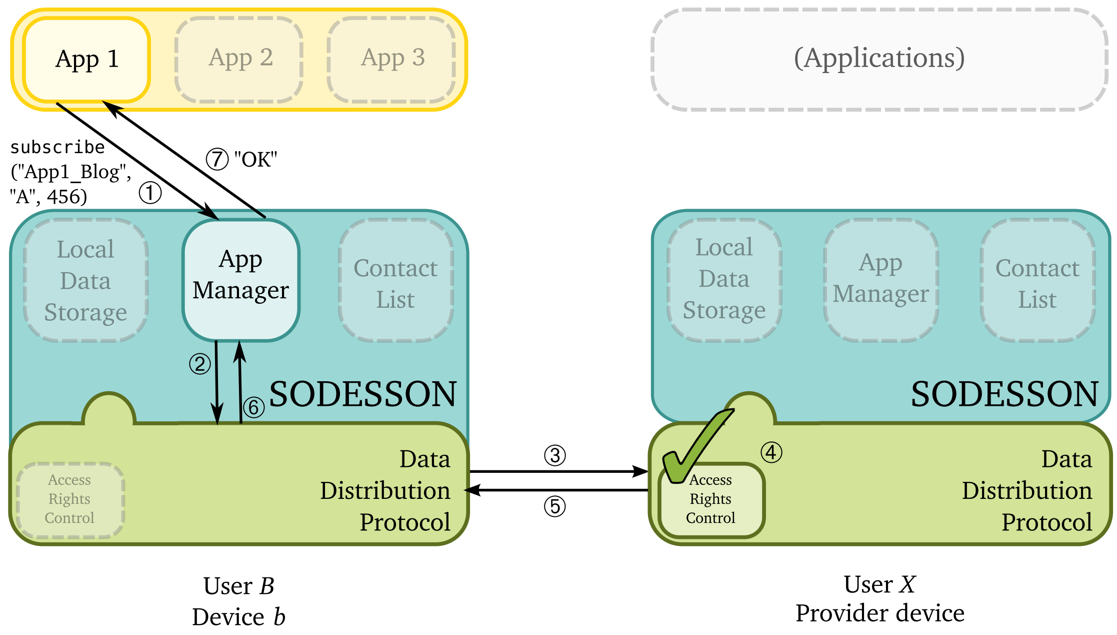


Figure 4.4: *Publish/subscribe, step three: subscription*

The App Manager prematurely stores the application's subscription. Next, the subscription request needs to be indicated at a provider device, as these are responsible for enforcing the access rights of this topic. Therefore, SODESSON passes the subscription request the DDP ②. The DDP forwards the subscription request to the correct provider device ③. The provider device checks if B is an allowed subscriber ④. This is the case (see Step two), hence the provider device updates its Access Control component and enters B as a subscriber.

The provider device now returns a response to device b to indicate that the subscription request was a success ⑤. This response is passed to the App Manager ⑥ which forwards it to App 1 ⑦.

4.3.3.4 Step four: Publishing a new data object and delivering it to the subscribers

An application which requests to publish a data object δ_t with topic t calls the method publish with the arguments as given in Table 4.6. The arguments are as follows:

Table 4.6: *Arguments for publish for publishing data object δ_t*

Data type	Argument	Description
`String`	topicTitle	Topic title $title_t$
`UID`	topicOwner	UID of the topic owner TO_t
`Integer`	objId	*(Optional:)* Object ID OID_{δ_t}
`BinaryData`	content	Content of data object
`Boolean`	inline	Data object is passed as inline value?
`Integer`	ttl	Time-to-live for δ_t

First, the topic $t = \{title_t, TO_t\}$ is passed via the arguments *topicTitle* and *topicOwner*.

The *objId* is an integer value for OID_{δ_t} to uniquely identify the data object. It is an optional argument: it can be passed by the application as a specific value in order to give δ_t a specific, application-internal, semantic meaning. For example, in order to update a data object that was earlier published, it must be addressed by the application via the same OID. Here, the application should generate OIDs on its own, pass them as an argument and keep track of them and the mapped data objects. See Section 4.3.7 about updating an existing data object. If the OID does not get passed by the application, the SODESSON middleware automatically creates a random value for it.

The *content* of the published data object is application-specific binary data. If the boolean argument *inline* is set to true, this indicates that the content is directly passed as the *content* argument itself. As an alternative (with *inline* set to false), an application can pass a local filepath as the *content* argument.

The argument *TTL* indicates a time-to-live in seconds for the data object δ_t, counting from the time where publish was called. Afterwards the TTL has been exceeded, the data object is seen as obsolete and must deleted from the Local Data Storages. A value 0 means a data object is never obsolete.

On receiving the publish method call, the SODESSON middleware stores the data object (either the inline content or the filepath reference, depending on the *inline* argument) in the Local Data Storage. The publish timestamp $time_{\delta_t}$ and the publisher's UID – both are parts of the data object's identifier as defined in Section 4.3.1 – get automatically set by SODESSON when it stores δ_t in the Local Data Storage.

The data object δ_t is then passed to the DDP: the identifier (topic title $title_t$, topic owner TO_t, Object ID OID_{δ_t}, publish timestamp $time_{\delta_t}$), as well as the publisher UID and the content / filepath reference.

The DDP now has the task to deliver δ_t (identifier and actual content) to all subscriber devices. To this end, two steps must be filfilled: first, a provider device must check its Access Rights Control whether the publishing user is an allowed publisher. Second, if the publisher is indeed allowed, the provider device stores δ_t in its Local Data Storage and deliver δ_t it to the subscriber devices. To this end, the provider device needs the required information which users are subscribers in its Access Rights Control module.

After the data object has been delivered to a subscriber device, SODESSON stores the data object in the Local Data Storage. During subscription, a mapping between the application's handle and the subscribed topic was stored in the App Manager module. Therefore, the application gets notified that a new data object has arrived for it. SODESSON calls the application interface method notifyApp with the arguments as displayed in Table 4.7.

During publish (see above), the arguments *topicTitle*, *topicOwner*, (optionally) *objId*, *content* and *inline* were set by the application. During notifyApp on the subscriber device, these arguments have an identical meaning and have identical values[3]. The value of *publishTime* $time_{\delta_t}$ was automatically set by SODESSON on the publishing device, as well as the *objId* if it was not passed

[3]The only exception here is the *content* if *inline* is false. In this case, the local filepath can differ from the one on the publishing device.

Table 4.7: *Arguments for notifyApp*

Data type	Argument	Description
`String`	topicTitle	Topic title $title_t$
`UID`	topicOwner	UID of the topic owner TO_t
`Integer`	objId	Object ID OID_{δ_t}, as set either by the publishing application or SODESSON on the publishing device
`Integer`	publishTime	Publish timestamp $time_{\delta_t}$, as set by the publishing device
`UID`	publisher	UID of the publisher P_{δ_t}
`BinaryData`	content	Content of data object δ_t
`Boolean`	inline	Data object is passed as inline value?
`Integer`	size	Inline content bytesize / filesize of data object δ_t

by the application. The field *size* holds the bytesize of either the inline content or the referenced file.

Example workflow

Figure 4.5 depicts the workflow of this step by continuing the private blog example.

Remember that in step one, user A bound the application "App 1" to SODESSON on device a. In step two, user A created the topic $t = \{\text{App1\_Blog}, A\}$ for his private blog and defined himself as an allowed publisher and B as an additional allowed subscriber. In step three, B subscribed to that topic. Now, a data object (blog post) created by A on device a shall be delivered to device b of subscriber B.

A creates a new blog post using App 1. App 1 publishes the blog post as a new data object, i.e. it calls the application interface method publish with the following arguments as given in Table 4.8.

Even though it is optional, the *objId* = 42 is set and memorized by the application to uniquely identify the blogpost. This way, existing blogposts can be updated later by publishing a new data

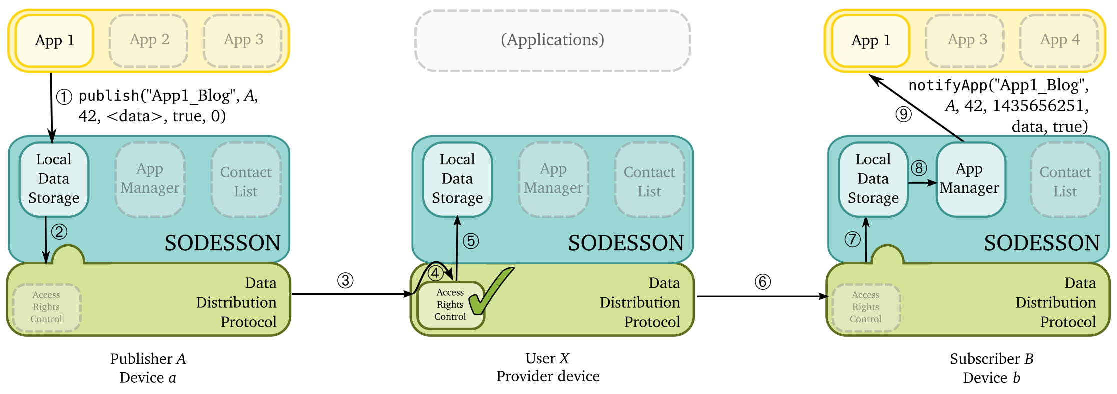


Figure 4.5: *Publish/subscribe, step four: publishing a new data object*

Table 4.8: *Arguments for publish (blog example)*

Data type	Description	Value in this example
`String`	topicTitle	App1_Blog
`UID`	topicOwner	UID_A
`Integer`	objId	42
`BinaryData`	content	<blogpost><title>We have a new dog!</ti...
`Boolean`	inline	true
`Integer`	ttl	0

object with the same *objId*. The blog application encodes a blog posts as an XML document (the *content*), which is here passed *inline*. The *ttl* is set to 0 because the blog post shall never expire.

On the publish method call, the SODESSON middleware stores the data object in the Local Data Storage of device *a* ①.

The blog post is then passed to the DDP ②: the passed identifier is {App1_Blog, *A*, 42, 143656251}, i.e. topic title, topic owner, objId and publish timestamp. The passed content is the blog post XML document.

After that, the blog post is forwarded to a provider device ③ which verifies *A*'s publishing rights ④. The provider device stores the data object in its own Local Data Storage ⑤ and delivers the blog post to *B*'s device *b* ⑥.

After the data object has been delivered at subscriber *B*'s device *b*, SODESSON on *b* stores it in the Local Data Storage ⑦. Due to the mapping between App 1's handle and the subscribed topic in the App Manager module ⑧, App 1 now gets gets internally notified that a new data object has arrived for it. SODESSON calls the application interface method notifyApp ⑨ with the arguments as shown in Table 4.9.

Table 4.9: *Arguments for notifyApp (blog example)*

Data type	Argument	Value in this example
`String`	topicTitle	"App1_Blog"
`UID`	topicOwner	UID_A
`Integer`	objId	42
`Integer`	publishTime	143656251
`UID`	publisher	UID_A
`BinaryData`	content	<blogpost><title>We have a new dog!</ti...
`Boolean`	inline	true
`Integer`	size	39765

4.3.3.5 Summary

These four steps have shown with the help of an example how a data object gets delivered from a publisher (user *A*) to a subscriber (user *B*). It was shown how *A* has defined the allowed publishers and allowed subscribers for a specific topic. Afterwards, *B* subscribed to said topic. Finally, *A* published a data object to said topic.

All steps relied on communication between *A*'s device a, *B*'s device b and one to be defined provider device. This inter-device communication is the task of the Data Distribution Protocol (DDP), which is undefined so far. With SocioPath, this thesis presents a DDP for SODESSON in Chapters 5 to 7.

4.3.4 Module: App Manager

During the four steps above, the App Manager module was used during two occasions. First, when an application registers and binds to SODESSON (step one). Second, when an application subscribes to a topic (step three).

The result of both processes – registration and subscription – are held locally in the App Manager's data structure. Each application is device-wide uniquely identified by a handle. With a mapping of handle and topic, SODESSON tracks which applications have to be notified by a `notifyApp` call when a data object associated to a subscribed topic is delivered to the device. Since multiple handles can be mapped to a given topic, the App Manger acts as a multiplexer: each data object with topic t is passed to every application subscribed to topic t.

The structure of App Manager is quite simple – each entry consists of three fields only (see Table 4.10):

Table 4.10: *Fields of an App Manager entry*

Data type	Field name	Description
`String`	topicTitle	Topic title $title_t$
`UID`	topicOwner	UID of the topic owner TO_t
`Array<Integer>`	handles	List of handles

A handle for an application is generated dynamically during the `registerApp` method and returned to the application. As soon as the application subscribes to a topic, it passes the topic title, topic owner and its handle to SODESSON via the `subscribe` method, as described in Section 4.3.3, step three. Subsequently, the entry for that topic gets created or updated, depending if there is already an entry for that topic and other handles.

4.3.5 Module: Local Data Storage

During the four steps for publishing, the Local Data Storage was mentioned multiple times. This section describes this module and its data structure in detail.

The Local Data Storage on a device holds all known information about a data object. A data object can either be stored as *inline content* within the data structure itself, i.e. by the binary data being filled into a specific entry's field, or it can be referenced as a file on the local filesystem. Both ways are available to an application during the publish call and the application can freely choose the method which suits the current use-case better. For example, for a filesharing application it would be inefficient to fully load an existing file and pass it inline to the Local Data Storage. On the other hand, for an instant messaging application it would result in unnecessary overhead to create a file for each small message and reference it during publish.

SODESSON supports the decoupling of *notifications* about a data object and *retrieval* of the actual data object content. This means that a subscriber device can be informed about the existence of a specific data object first before the actual content is delivered to that device. As a result, the Local Data Storage can hold an entry about a data object, but neither inline content nor a filepath reference exists. Notifications and delivery decoupling is further discussed in Section 4.3.6.

For each published data object, a Local Data Storage entry can exist for three different device roles:

(1) **Publishing device:** On publish by an application, the publishing device stores or references the data object in the Local Data Storage.

(2) **Provider device:** A provider device which shall deliver a data object to subscriber devices stores or references this data object in its Local Data Storage.

(3) **Subscriber device:** After a successful delivery of the data object to a subscriber device, the subscriber device's Local Data Storage holds or references the data object and makes it available for the applications.

Table 4.11 shows the fields of a Local Data Storage entry for a data object δ_t.

Table 4.11: *Fields of a Local Data Storage entry for data object δ_t*

Data type	Field name	Description	Optional?
`String`	topicTitle	Topic title $title_t$	
`UID`	topicOwner	Topic owner TO_t	
`Integer`	objId	Object ID OID_{δ_t}	
`Integer`	publishTime	Publish timestamp $time_{\delta_t}$	
`UID`	publisher	UID of the publisher P_{δ_t}	
`Integer`	gotNotifyTime	Timestamp of notification delivery	yes
`Integer`	ttl	Time-to-live TTL_{δ_t}	
`BinaryData`	content	Inline content of δ_t	yes
`String`	filepath	Local filepath reference to δ_t	yes
`Integer`	size	Inline content bytesize / filesize of δ_t	
`Array<DID>`	sources	Known devices that hold δ_t	

The fields *topicTitle* and *topicOwner* define the data object's topic. Together with the topic, *objId* and *publishTime* uniquely identify the data object, as defined in Section 4.3.1. Additionally, the *publisher* UID is stored, so the data object's publisher can be identified.

The field *gotNotifyTime* indicates the time when a device has learned for the first time about a specific data object, i.e. received a notification or the full data object. This field is optional. Its information is not mandatory for the publish/subscribe service to work, but could contain helpful additional information for the DDP, for example SocioPath makes use of this timestamp for synchronizing notifications between devices. This timestamp is set automatically by SODESSON on notification or data object delivery.

The *time-to-live* TTL_{δ_t} is passed during the SODESSON method call publish. It is counted in seconds from $time_{\delta_t}$ onwards. After exceeding the time-to-live, the entry is pruned from the Local Data Storage by a periodically executed cleanup task. If the application sets TTL_{δ_t} to 0 during publish, δ_t is never obsolete.

Depending on whether a data object is stored inline or referenced as a file, either the field *content* or *filepath* is set with the according value. At most one of the two value can be set, i.e. both fields are mutually exclusive. If a device holds only a notification, but not the data object's content yet, both fields are empty.

The field *size* holds the bytesize of δ_t's content, i.e. either the size of the inline content or the size of the referenced file.

The fields *sources* holds a list of other devices which are known to hold δ_t. Each device is identified by its globally unique Device ID (DID) as defined by the DDP. Since SODESSON is device-agnostic, this field can be freely filled by the DDP in order to keep any provider device information for the respective data object entry. It is up to the DDP to fill it with relevant information and keep it up-to-date. This list is used to retrieve δ_t at a later time if the Local Data Storage entry is only a notification (see Section 4.3.6).

An example Local Data Storage entry for the given blog post example is displayed in the next section, in table 4.13.

4.3.6 Retrieving a Data Object

SODESSON supports the decoupling of **notifications** about and **retrieval** of the actual data object. This means that a subscriber device can be informed about the existence of a specific data object first before the actual content is delivered to that device.

The reason for this possibility is SODESSON's support of resource awareness. A data object can have an arbitrary bytesize and therefore be multiple mega- or gigabytes large. Delivering such a data object unsolicitedly to a subscriber device that has limited data storage or a metered network connectivity is not preferrable (see Section 3.2.3). Instead, it is more flexible to send a small piece of information, i.e. a notification, to the subscriber device first to inform it about the data object. The notification contains the data object's metadata, i.e. the identifier (topic, objId, publish timestamp) and the content's bytesize. The subscriber device can now decide if and when it wants to retrieve the **full data object**, i.e. the data object's metadata plus content.

The notification is basically a data object without the content. When the notification arrives at a subscriber device, the same workflow as on data object delivery is executed (see Section

4.3.3.4): the application receives a notifyApp call where the *content* field is empty and the *inline* field is set to false.

As soon as the application decides to retrieve the data object's content, it calls the `retrieve` method with the arguments as given in Table 4.12.

Table 4.12: *Arguments for* `retrieve`

Data type	Argument	Description
`String`	topicTitle	Topic title $title_t$
`UID`	topicOwner	UID of the topic owner TO_t
`Integer`	objId	Object ID OID_{δ_t}, as set either by the publishing application or SODESSON on the publishing device
`Integer`	publishTime	Publish timestamp $time_{\delta_t}$, as set by the publishing device

The application fills these arguments with the values it has learned earlier from the middleware's notifyApp call. SODESSON now makes a best effort to retrieve the data object's content from provider devices via the DDP.

As soon as the data object's content has been delivered to the device, i.e. the full data object is stored in the Local Data Storage, SODESSON calls the notifyApp method again, this time with a non-empty *content* argument.

Whenever an application calls the `retrieve` message, it is transparent to the application if the full data object content is already available on the same device or if it has to be delivered by a provider device first. The middleware sends a notifyApp with an non-empty "content" argument as soon as the full data object is stored in the Local Data Storage.

Deciding when to retrieve

The decision can be made either on application or DDP level. When the retrieval is triggered by an application, e.g. by user interaction, it shall immediately call `retrieve`, as described above. Note that the notifyApp call contains the size of the data object, thus the decision can be made based on the size.

The DDP could implement a resource-aware intelligence based on the device's connectivity, storage etc. and autonomously decide to retrieve the data object's content – independent from any `retrieve` calls by the application. In that case, the data object the middleware sends a notifyApp with an non-empty *content* argument before the application decides to call `retrieve`.

As a prerequisite, this decoupling needs to be supported by the DDP, i.e. the DDP needs to provide the possibility to create a notification from the data object in the Local Data Storage of a provider device, deliver the notification to the subscriber device, return a the retrieval request to a provider device and only then deliver the data object's to the subscriber device. SocioPath is a DDP which supports the decoupling of notifications and data object retrievals. This is explained in Section 5.7.

Workflow

Figure 4.6 displays the workflow based on an example. It revisits the example from Section 4.3.3 where a blog post was published.

Assume that a new blog post ϵ_t with Object ID $OID_{\epsilon_t} = 73$ is filled with high resolution photos and about 100 megabytes in bytesize, therefore it is reasonable to decouple notification and the data object content. Hence, a provider device delivers a notification to subscriber device b first ①– ④ (this is analogous to the ⑥– ⑨ in Section 4.3.3.4). This notification is stored as an entry in the Local Data Storage ② with the values shown in Table 4.13.

Table 4.13: *Local Data Storage entry for the example blog post*

topicTitle	topicOwner	objId	publishTime	publisher	GotNotify Timestamp
App1_Blog	UID_A	73	143680191	UID_A	143690111
TTL	Inline Data	Filepath	Bytesize	Sources	
0	NULL	NULL	100123814	DID_x	

Note that both inline data and filepath fields are empty (NULL).

The application now decides to retrieve the blog post, possibly by user interaction ⑤. This request is passed via the Local Data Storage, where the known sources are gathered, to the DDP ⑥. As soon, as the retrieve request reaches a provider device ⑦, the to-be-retrieved data object is extracted from the Local Data Storage ⑧ and passed to the requesting device ⑨.

Not depicted in Figure 4.6 are the final steps on the subscriber device where the notifyApp method is eventually called by the App Manager. These steps are identical to the steps ⑦ to ⑨ in Section 4.3.3.4.

4.3.7 Updating and deleting a data object

SODESSON gives applications the ability to update (overwrite) and delete data objects that have been published before. Both actions are essentially published data objects, therefore the workflow for delivering an updated data object or indicating a deletion to the subscriber device is identical to a new publish, as described in Section 4.3.3, step four. Furthermore, the same access control rules apply: only allowed publishers may update or delete an existing data object for a given topic.

Updating

Let δ_t be an existing data object which is uniquely identified by the following 3-tuple (see Section 4.3.1):

- Topic t
- Object ID OID_{δ_t}
- Creation timestamp $time_{\delta_t}$

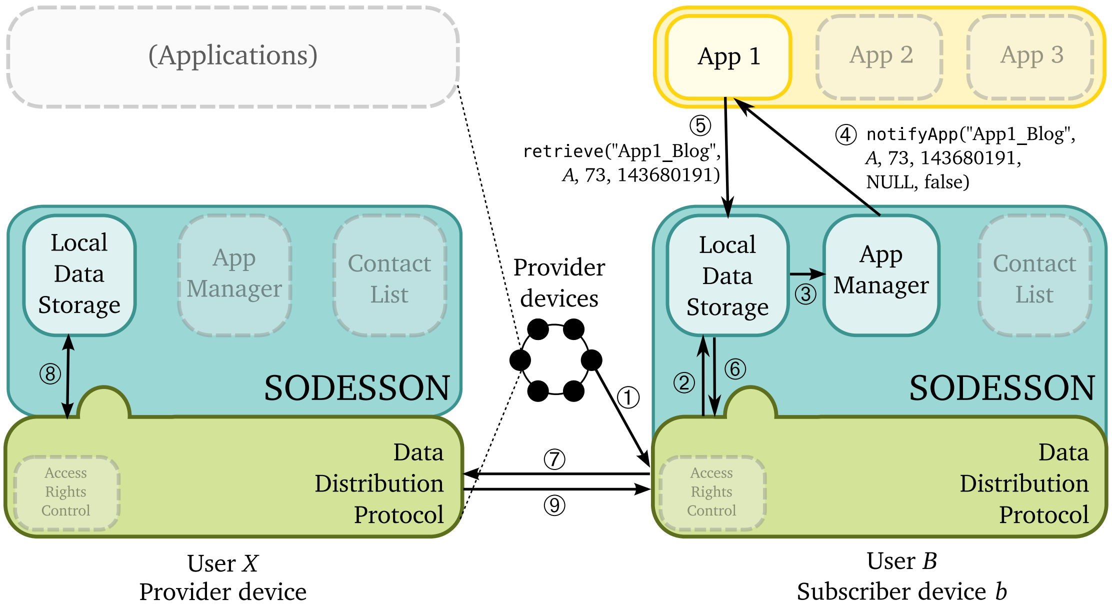


Figure 4.6: *Notification about and retrieval of a data object*

In order to update δ_t to δ'_t, an allowed publisher has to publish a new data object δ'_t with the following identifier values:

- Topic t
- Object ID OID_{δ_t}
- Creation timestamp $time_{\delta'_t} > time_{\delta_t}$

Thus, the updated data object δ'_t has the same topic and OID as the original data object δ_t, but a newer creation timestamp. On delivery to a subscriber device, SODESSON checks if information about an existing data object topic with t and OID_{δ_t} is stored in the Local Data Storage. If true and if the creation timestamp is newer than the timestamp of the formerly stored data object, the entry is overwritten by the new data object. If false, the "new" data object simply is stored.

In order for this mechanism to work, the clocks of the participating devices for measuring timestamps have to be synchronized, as assumed earlier in Section 4.3.1.

Deleting

Deletion of an existing data object is basically an update with empty inline content. Let δ_t be an existing data object which is uniquely identified by the following 3-tuple (see Section 4.3.1):

- Topic t
- Object ID OID_{δ_t}
- Creation timestamp $time_{\delta_t}$

In order to delete δ_t, an allowed publisher has to publish a new data object δ'_t with the following identifier values:

- Topic t
- Object ID OID_{δ_t}
- Creation timestamp $time_{\delta'_t} > time_{\delta_t}$
- content: NULL
- inline: true

On subscriber devices, SODESSON checks if an entry for an existing data object with topic t and OID_{δ_t} is stored in the Local Data Storage. If true, SODESSON deletes said entry.

On provider devices, the entry must not be deleted from the Local Data Storage since the provider device must deliver the information about the deletion to the subscriber devices. Still, the provider device can at least delete the content of the data object and keep the remaining metadata.

4.4 Contact management

On each device, SODESSON holds a common **Contact List** which is available to all applications via the getContacts method. This eliminates the need for each U2U application to implement their own handling of contacts. A topic owner uses his devices' Contact List entries to define the sets of allowed publisher and allowed subscribers for each topic t he owns, as discussed in Section 4.3.3.2. Each user has exactly one contact list, which the DDP must replicate to all devices of this user.

4.4.1 Data structure: Contact List

The Contact List on a device of user A contains $\mathbb{C}_A$ and A himself. An entry for user X in this Contact List has the fields as shown in Table 4.14.

Table 4.14: *Fields of a Contact List entry*

Data type	Field name	Description
`UID`	userId	User ID UID_X as defined in Section 4.3.1
`String`	alias	Human readable alias, nickname, etc. Arbitrary string.
`String`	publicKey	Public key $pubkey_X$, as as defined in Section 4.3.1
`String`	privateKey	Private key $privkey_X$. This value is only set for $X = A$.

4.4.2 Initial Setup

For each user, the SODESSON middleware gets initialized on a first device. Here, an application, e.g. a setup wizard, leads the user through the setup and calls the `registerUser` method to create a public/private keypair and initialize an empty App Manager and a Contact List which only holds the newly registered user himself: his public / private key and his *UID* as a hash of the public key. An alias can be set via the method `editContactAlias` (see Section 4.4.4).

For each additional device of the same user, the keypair needs to be imported on the new device via the `importUser` method. Here, the keypair has to be transferred to the new device's filesystem first. The SODESSON middleware does not cover a method in its interface, so it has to be done manually, e.g. by copying the files into a pre-defined directory.

The two methods `registerUser` and `importUser` are explained in detail in Appendix B.

4.4.3 Adding contacts

A new contact can be added to the Contact List via an contact management application which calls the `addContact` method with the arguments as shown in Table 4.15.

Table 4.15: *Fields of a Contact List entry*

Data type	Argument	Description
`String`	publicKeyFilepath	This path to a file tells SODESSON where to find the contact's public key file.
`String`	alias	*(Optional:)* User alias for better human readability.

With these arguments, an entry in the Contact List is created, with the *UID* being the hash of the public key, as defined in Section 4.3.1. After a new contact has been added to SODESSON's Contact List, the DDP is notified about this event via an internal hook, so it can perform any device-level maintenance that may be required due to the new contact.

Public key exchange

Before user A with device a can call the `addContact` method for user B, the public key of B has to be stored on a first.

To this end, it is assumed that the two users have exchanged their public keys and verified their identity earlier over a secure channel. This procedure happens out-of-band, independently from SODESSON. One example for such a procedure is described in [26]: here two devices a and b of two users A and B first locally broadcast *discovery* messages. The message from device a contains a user alias for A and a's IP address, the message from device b contains analogous values. Then A and B mutually agree to an *introduction phase* where their public keys are sent to the other device, using the IP addresses learned from the discovery message. Finally, A and B enter a *verification phase*, where they verify each other's connection of the public key to the user's identity, e.g. by reading aloud the public keys fingerprint. After these three phases, the users A and B have the other's public key on their device and verified its owner's identity. The procedure on either device can now instantly call SODESSON's addContact method with the alias from the discovery message and the verified public key. This finalizes the key exchange for SODESSON.

4.4.4 Editing contacts

A user can remove a contact from his contact list. This is done by the method removeContact. In this case, similar to updateTopic, the responsible provider devices gets notified to remove the deleted contact as allowed publisher, allowed subscriber and subscriber from all topics the calling user is owner of.

A user can also edit a contact's alias via the method editContactAlias.

The two methods removeContact and editContactAlias are defined in Appendix B.

4.5 Conclusion

This chapter presented SODESSON, a middleware for enabling U2U communication applications. SODESSON handles U2U communication as defined in Section 2.1, i.e. it enables publishing data objects and subscribing to topics. Besides topics, SODESSON complies with the U2U concepts of topic owners, allowed publishers and allowed subscribers.

Section 4.2 described SODESSON's basic architecture, including how the U2U entities user, topic and data object are addressed.

Section 4.3 described SODESSON's topic-based publish/subscribe service for applications, i.e. the service's interface methods and modules. This service allows to define allowed publishers and allowed subscribers, to subscribe to topics and publish data objects for a given topic. Four steps for publishing a data object and the respective interface methods were explained with the help of the example of publishing a blog post. Besides publishing new data objects, updating and deleting existing data objects was explained. Additionally, SODESSON's support decoupling of notifications and data object retrieval was introduced. This decoupling plays an important role for establishing resource awareness later.

Section 4.4 described SODESSON's contact management, i.e. the addition and editing of contacts. A topic owner selects allowed publishers and allowed subscribers from his contacts.

Devices are the only U2U entities that were not covered so far. SODESSON by itself is agnostic of any device handling and provider schemes. It can leverage third-party provider schemes as well as User-Centric Networking. To this end, SODESSON needs to be complemented with a Data Distribution Protocol (DDP) which handles the inter-device communication. In order to realize User-Centric Networking, this thesis presents SocioPath in the upcoming chapters – a self-sufficient, partition-tolerant and resource-aware DDP for SODESSON.

Chapter 5

SocioPath: Protocol Overview

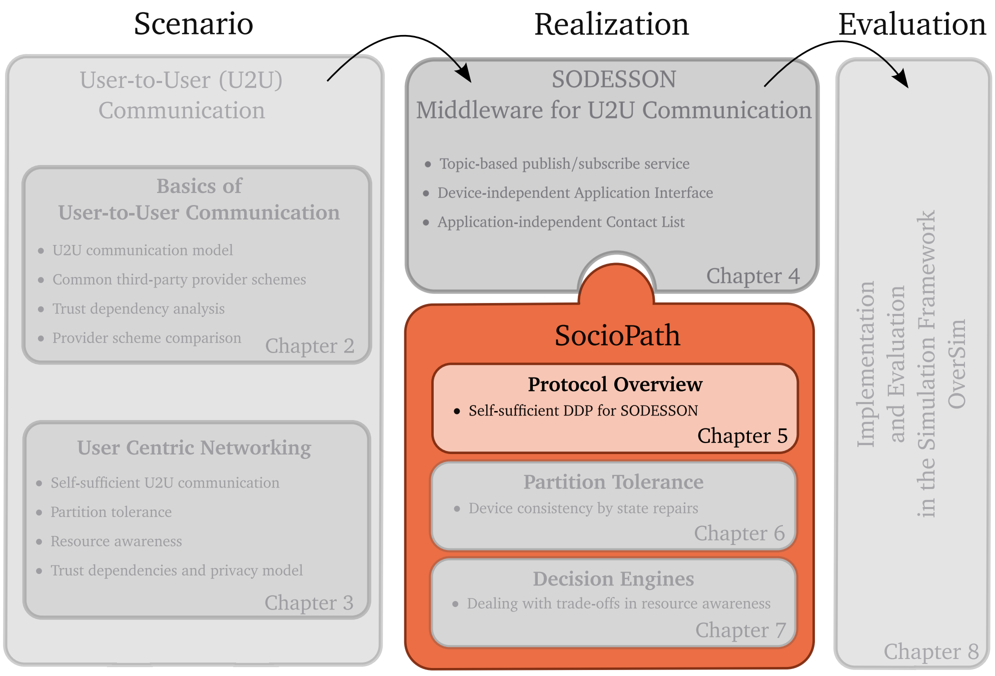

Scenario
Realization
Evaluation
User-to-User (U2U) Communication
Basics of User-to-User Communication
U2U communication model
Common third-party provider schemes
Trust dependency analysis
Provider scheme comparison
Chapter 2
User Centric Networking
Self-sufficient U2U communication
Partition tolerance
Resource awareness
Trust dependencies and privacy model
Chapter 3
SODESSON Middleware for U2U Communication
Topic-based publish/subscribe service
Device-independent Application Interface
Application-independent Contact List
Chapter 4
SocioPath
Protocol Overview
Self-sufficient DDP for SODESSON
Chapter 5
Partition Tolerance
Device consistency by state repairs
Chapter 6
Decision Engines
Dealing with trade-offs in resource awareness
Chapter 7
Implementation and Evaluation in the Simulation Framework OverSim
Chapter 8

Figure 5.1: *Placement of Chapter 5 in the big picture*

5.1 Overview

In Chapter 3, User-Centric Networking was presented as a self-sufficient, partition tolerant and resource-aware approach for U2U communication. The presented advantages in User-Centric Networking are better privacy (through self-sufficiency) and considering the demands of user device's compared to other decentralized provider schemes: partition tolerance enables U2U communication in spite of failing and changing network connectivities, while resource awareness takes each device's availability and capability into account.

However, the concept of User-Centric Networking still needs to be realized. Chapter 4 presented the SODESSON middleware which is a topic-based publish/subscribe service for U2U communication that provides a generic application interface, abstraction from devices and application-independent contact list. SODESSON must be complemented by an exchangable *Data Distribution Protocol (DDP)*, which handles inter-device communication to fulfill two tasks: first, enforcing the access rights, i.e. making sure that only data objects that were published by allowed publishers get delivered to the subscribers (and only these). Second, delivering the data object itself. To this end, two tasks are relevant for a DDP and responsible provider devices must be defined by it. First: controlling and enforcing access rights, second: storing and delivering data objects to subscriber devices. Based on the roles of publishing-, provider- and subscriber devices, the DDP must define its protocol for data delivery.

SocioPath is a self-sufficient DDP for SODESSON to enable User-Centric Networking as the basis for U2U communication. In the following, an overview is given how the three aspects of User-Centric Networking get realized. The details are discussed throughout the upcoming chapters.

- **Self-sufficiency**: As described in Section 3.1, self-sufficiency means that only devices of the closed group's members ever get in touch with a published data object: publishing device, topic owner's devices and subscriber devices. This increases the members' privacy at least to privacy level III (cf. the privacy model introduced in Section 3.6).

  Since SocioPath is a self-sufficient DDP, privacy level III is already an improvement in user privacy compared to storage schemes which involve third parties. Depending on the used Decision Engine (see below), even privacy level IV can be achieved.

  In SocioPath, only the topic owner's devices perform access control, since in User-Centric Networking only the topic owner's devices hold the information about allowed publishers, allowed subscribers and subscribers (see Section 3.1.1). Therefore, published data objects cannot get delivered to the subscriber's devices unless the topic owner forwards the required information (i.e. the data object itself or a notification with a sources list) to other users.

- **Partition tolerance through state repairs**: Sometimes user devices are switched off or lack availability for other devices due to missing network connectivity. During these times, a device misses notifications about new or updated data objects. SocioPath deals with this problem with state repairs: At certain times, two devices exchange their states and repair possible inconsistencies. Until the repair, each device works on its own local view and fully participates in SocioPath communication: state inconsistencies are not seen as fatal, but accepted as a regular aspect.

  This makes SocioPath especially resilient to partitions which are possible in a UCN. For example, if two devices a and b only communicate in a local network (Partition 1), but only a later gains Internet access and can communicate with another device c (Partition 2), c can gain the same state as a and b, even with b not being part of Partition 2.

- **Resource awareness through exchangable Decision Engines**: Section 1.4.3 described the trade-offs in resource awareness. Three preferable goals were identified that are mutually exclusive: resource conservation, low delays and group members' privacy. This trade-off offers different preferable policies, depending on the available devices and application use-case.

  Greedy automatic retrieval of a very large data object results in low access delays for the user, if it is already downloaded to the device by the time the user wishes to access the data object. However, such a policy may come at high communication costs for the user, for example if the policy does not take into account whether the device currently has mobile or WiFi reception. On the other hand, lazy retrieval which always has to be user-demanded gives the user full control over the costs, but may result in higher access delays for him.

  Likewise, resourceful devices of a subscriber can be preferable storage devices for other subscribers. However, if privacy level IV shall be reached, this is not an option, since information about other subscribers is passed from the topic owner to another user. This results in privacy level III.

  Depending on the device distribution, and with regard to resources and the overall average number of devices per user, one policy might be more preferable than the other.

  Therefore, instead of fix protocol behaviour, SocioPath specifies a set of *events* and *actions* with demanded outcomes (e.g. "notify all subscriber devices"). An exchangeable and expandable *Decision Engine (DE)* freely defines how these events are handled and how flexible actions are performed in order to achieve the demanded outcomes.

5.1.1 Outline of the upcoming sections and chapters

The description of the SocioPath protocol is split across next three chapters as follows:

The remainder of this chapter deals with SocioPath's protocol design. First, SocioPath's internal data structures are explained, which are required for communication between two devices and access rights enforcement. Afterwards, the fundamentals of data object delivery are explained. Next, the details of the protocol flow are described, including with the help of the SODESSON blog post example, which will be revisited. Section 5.6 explains additional maintenance mechanisms. Section 5.7 deals with the decoupling of notifications and data object retrievals and and explains how large data objects can be distributed between devices efficiently.

Chapter 6 explains how partition tolerance – one central aspect of User-Centric Networking – is achieved. Here, consistency between device is reached by state repairs.

Chapter 7 discusses the tasks of a Decision Engine. Here, a workflow is presented with well-defined events and actions. Each Decision Engine takes this workflow as a template, and specifies its behaviour on these well-defined events and action. As an example, the three DEs *Instant-to-All, Offload-First* and *Helping-Friends* are presented, where each aims at a trade-off and cover two of the three mutually exclusive goals of resource awareness.

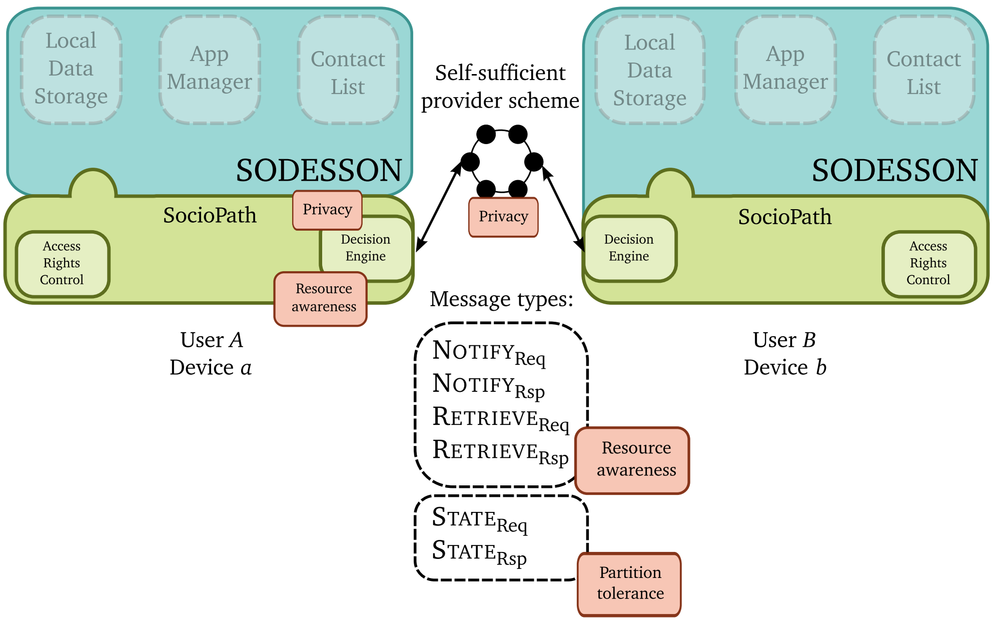


Figure 5.2: *Integration of SocioPath into SODESSON*

5.2 Internal Data Structures

Before the exchange of data objects in SocioPath is explained, the data structures maintained locally on each SocioPath device have to be introduced first. This section describes these data structures. Note that all symbols in the descriptions were defined in Section 4.3.1.

5.2.1 Devices List

The Devices List of user A holds information about all of A's and A's contacts' devices. This Devices List gets replicated on all of A's devices. As a consequence, only devices of the same user or mutual contacts can communicate directly with each other.

Each device a has a unique identificator DID_a which is a UUID [66]. DID_a is generated during initial setup. It is used to address a device in SocioPath in a location-independent way, since a locator such as a IP address and port might change over time.

Additionally, each device's last known Communication Costs Value (CCV) is stored. The CCV is an abstract value for indicating a device's capability and can be interpreted by resource-aware Decision Engines. Each device sends its current CCV to other devices in every SocioPath message, as will be discussed in Section 5.4.

Hence, an entry for device b of user B in the Devices List on one of A's devices has the fields shown in Table 5.1.

Table 5.1: *Fields of an Devices List entry*

Data type	Field name	Description
`UID`	userId	UID_B, as defined in Section 4.3.1, who controls device b
`DID`	deviceId	DID_b of the device b
`Locator`	deviceLocator	Underlay network locator, e.g. IPv6 address and UDP port
`Integer`	ccv	Last known Communication Costs Value (CCV)

5.2.2 Own Topics List

The Own Topics List on a device of user X holds information about all topics X is owner of. Each topic t in this list is identified by its title $title_t$ only, since the topic owner TO_t is trivially X himself.

The access rights per topic t are defined by three lists which hold the *UID*s of $\mathbb{P}_t$, $\mathbb{A}_t$ and $\mathbb{S}_t$ respectively.

X can change $\mathbb{P}_t$ and $\mathbb{A}_t$ on any device he controls and changes get synchronized with his other devices by publishing the respective data. When the corresponding data object arrives at the other topic owner devices, these check if the data object's timestamp is newer to the *lastUpdate* timestamp. This ensures sure that obsolete changes that arrive at the device delayed do not overwrite newer entries. The *lastUpdate* timestamp is updated accordingly.

As discussed in Section 4.3.3.2, the application can optionally pass a targeted privacy level, according to the privacy model in Section 3.6.2.

Hence, an entry for topic t has the fields displayed in Table 5.2.

Table 5.2: *Fields of an Own Topics List entry*

Data type	Field name	Description
`String`	topicTitle	Topic title $title_t$
`Array<UID>`	allowedPubs	UID list of allowed publishers $\mathbb{P}$
`Array<UID>`	allowedSubs	UID list of allowed subscribers $\mathbb{A}$
`Array<UID>`	subscribers	UID list of subscribers $\mathbb{S}$
`Enum`	privacyLevel	*(Optional:)* Privacy level as defined by the application
`Integer`	lastUpdate	Last update timestamp

The Own Topics List is replicated on each device of user X.

5.2.3 Subscriptions List

The Subscriptions List on a device of user A holds information about all topics A is subscribed to. This list includes all topics A is topic owner of, as well as topics that other users own. The Subscriptions List acts as a filter for incoming data objects to decide whether the data object is relevant for the user and gets entered into the Local Data Storage. More importantly, this information is required for STATE exchanges which is a building block to partition tolerance (see Chapter 7).

An entry for topic t in the Subscriptions List has the following fields:

Table 5.3: *Fields of a Subscriptions List entry*

Data type	Field name	Description
`String`	topicTitle	Topic title $title_t$
`UID`	topicOwner	UID_{TO_t}

Note that the Subscriptions List is different from the App Manager in SODESSON. The Subscriptions List is synchronized across all devices of the same user and is independent from running applications on the device. The App Manager on the other hand holds a mapping between applications and topics via the application handles. Such a mapping is created / deleted via the application interface methods registerApp / unregisterApp.

5.3 Fundamentals of Data Object Delivery

In this section, the fundamentals for data object delivery in SocioPath are described. These consist of the following points:

- **Basic delivery process**: a short overview on how a data object is delivered. This will be further elaborated in the sections about implementation (Section 5.4) and an example workflow (Section 5.5).
- **Maintenance topics**: maintenance communication between devices in SocioPath, such as requesting a new subscription or introducing a new device into the UCN, is handled in the same way as a data objects published by an application. The maintenance information is also a data object and the users of the devices which require that information, are subscribers of a specific maintenance topic.
- **Encryption**: a sketch about how the information that SocioPath devices hold can be used for encrypted transmission via unsecure networks.

5.3.1 Basic delivery process

Initially, a data object gets created by an application on publishing device a_1 and passed down to the SODESSON middleware via the application interface (see Section 4.3). In SocioPath, each published data object δ_t shall be delivered to the full device pool $\mathbb{D}_{\delta_t}$ (see Section 3.1 for the notation), i.e. the following devices:

- all devices of the publisher
- all devices of the topic owner

- all devices of all subscribers

The list of subscribers is held in the Own Topics List of the topic owner's devices. Thus, a published data object first needs to be delivered to at least one topic owner device before it can be delivered to the subscribers, i.e. a topic owner device needs to forward (and thus deliver) the data objects to subscriber devices.

As soon as a device b of user B receives a data object, it checks for the following requirements:

- **Publishing rights**: (only if B is topic owner) The data object's publisher must be an allowed publisher, according to the Own Topics List on b
- **Relevance**: B must be either a subscriber of the data object's topic (according to the Subscriptions List on b) or a topic owner.
- **Non-Obsoleteness**: if the notification already exists in the Local Data Storage (according to topic and object ID), the publish timestamp of the incoming data object gets checked. If that timestamp is older than the timestamp of the notification that already exists in the Local Data Storage, the incoming notification gets discarded.

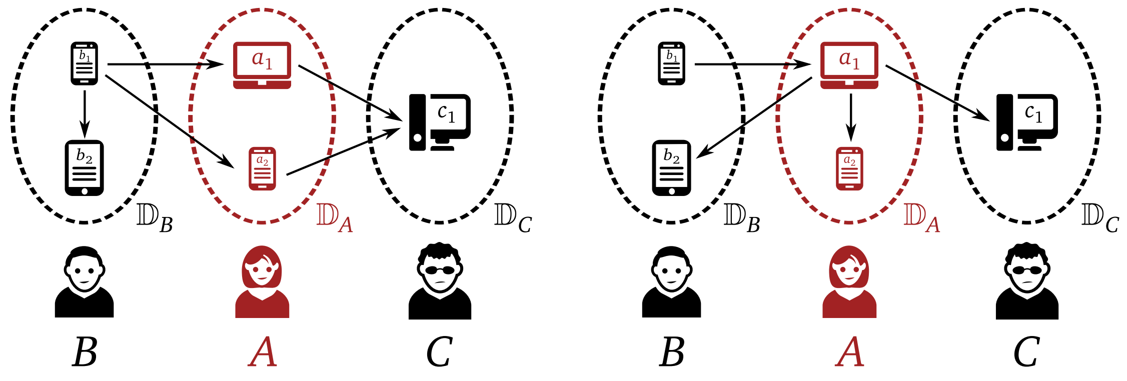
If all three requirements are fulfilled, the data object is entered into Local Data Storage if no entry with the respective topic and object ID exists yet. If it already exists and the publish timestamp of the data object is newer than the existing one's, the existing entry gets updated with the new data.

Note that the data object does not only get delivered to subscribers, but to all devices of all closed group members: publisher, topic owner and subscribers. All of these store the data object into their Local Data Storage. This is a straightforward approach to achieve redundancy, since only these devices are eligible as provider devices for self-sufficient U2U communication. By doing so, all these devices can deliver missed data objects to other formerly unavailable devices during STATE exchanges (see Chapter 6).

Since a data object shall be delivered to all of a user's devices, it has to be decided *which device sends the* NOTIFY$_{Req}$ *message to which device*. Solving this resource-driven problem is the task of the resource-aware Decision Engine. Similarly, if multiple devices already hold a data object, they are a potential source for delivering the data object to another device.

Figure 5.3 shows an example how data object delivery can be solved in two different ways. Assume that A is the topic owner, B is an allowed publisher and C is a subscriber. b_1 now publishes a new data object. In Figure 5.3a, b_1 sends the NOTIFY$_{\text{Req}}$ message to a_1, a_2 and b_2 by itself. b_1 cannot send the NOTIFY$_{\text{Req}}$ message to c_1 since B is not the topic owner and thus does not know that C is a subscriber. Instead, both a_1 and a_2 as topic owner devices each send the NOTIFY$_{\text{Req}}$ message to c_1. This is a unnecessarily redundant, but simple approach, since no coordination between a_1 and a_2 is done here.

In Figure 5.3b, a more resource-aware policy is used. Assume that a_1 has more resources than b_1. b_1 offloads the NOTIFY$_{\text{Req}}$ message to a_1, whereas a_1 sends the NOTIFY$_{\text{Req}}$ message to the remaining devices.



(a) Device b_1 sends NOTIFY$_{\text{Req}}$ messages to all publisher and topic owner devices by itself

(b) Device b_1 sends NOTIFY$_{\text{Req}}$ message to a_1 only, a_1 sends NOTIFY$_{\text{Req}}$ message to the remaining devices

Figure 5.3: *Two possibilities for delivering a data object*

Data transportation

SocioPath is an overlay protocol on ISO/OSI Layer 7. Due to the request-response communication pattern, it offers an acknowledged service: if the sending device of the request receives a response, it knows that the request message arrived. The sending device of the response does not directly know if it arrived.

There are no direct retransmissions. Instead, lost messages are handled by state repairs (see Chapter 6).

In its current design, SocioPath is agnostic of the underlying transport protocol. It merely assumes that there is a generic device locator, e.g. IPv6 address and TCP port. There is no defined upper limit for message sizes.

5.3.2 Maintenance Topics

Maintenance topics are used to keep the data structures Contact Lists, Device Lists, Own Topics Lists, and Subscriptions Lists up-to-date, since these have to adapt to constant changes: For example, if a user subscribes to a topic, all of his devices have to update the Subscriptions List and all the topic owner's devices have to update their Own Topics Lists. If two users add each other as a new contact, the devices of both users have to update their Contact Lists, and so on.

Maintenance topics are a result of re-using SODESSON's publish/subscribe mechanism for distributing maintenance updates via the DDP. The only difference is that data objects are not created by an application but internally by SocioPath itself. It is a straightforward solution to use the very same mechanisms as for new data objects from applications. Therefore, maintenance data objects also get stored in the Local Data Storage.

For each use-case that affects maintenance data structures, suitable maintenance topics are defined on a per user-basis (e.g. $t_1 = \{\text{"OwnTopics"}, A\}$, $t_2 = \{\text{"OwnTopics"}, B\}, \ldots$) and the access rights are set accordingly: For example, changes in the Own Topics List are only meant for the topic owner himself. Adding a new device is also relevant information for all his contacts, so a new data object published by one of these contacts reaches the new device as well.

Depending on the maintenance topic, updates are only for the topic owner himself or for the topic owner and his contacts. In the latter case, these contacts are subscribers for the respective maintenance topic. The topic owner's Own Topics List gets filled accordingly.

5.3.3 Encryption

Due to self-sufficiency, each data object only touches provider devices that are controlled by users that are allowed to read that data object. However, transmission of the data object may still take place via unsecure networks. On the networking layer, end-to-end encryption is still required to keep the data object confidential from third parties.

By the combination of channel encryption and self-sufficiency, content data confidentiality (see Section 2.3.1) for the closed group can be established: channel encryption achieves that infrastructure providers cannot read any data objects during delivery. Self-sufficiency achieves that third-party application providers cannot read any data objects, because third-party application providers are not involved in the delivery process.

In SocioPath, only devices of contacts are held in the Devices List, hence only devices of contacts communicate with each other. Furthermore, every device holds the private and public key of the owner and the contacts' public keys in the Contact List. This makes all encryption processes straightforward and besides content data confidentiality, content data integrity can be achieved. Given that the following prerequisites are fulfilled for two devices a and b of two contacts A and B:

- a and b can communicate with each other (via SocioPath's Device List)
- a (b) holds B's (A's) public key (via SODESSON's Contact List)
- a (b) holds A's (B's) private key (via SODESSON's Contact List)

Then a and b are able to perform asymmetrically encrypted communication on behalf of A and B. For performance reasons, a and b could also use their public key infrastructure to negotiate a symmetric key. Establishing the actual encryption process under these prerequisites is a solved problem by proven state-of-the-art security protocols. This issue is not further discussed here. For example, in order to secure a communication channel between two devices, Transport Layer Security (TLS) [23] or Datagram Transport Layer Security (DTLS) [83] can be used.

5.4 Protocol Flow Details

This section covers technical details about the protocol flow and most important message types in SocioPath that are required for successful data object delivery. These details are the basis for a later implementation.

5.4.1 General properties of messages

All messages in SocioPath are exchanged in a request-response pattern between two devices that hold each other in their Devices Lists. This means that each unsolicited message (request) from one device *a* to another device *b* shall be answered by *b* with a corresponding response. Each request message contains a nonce field which is filled by the sender with a random integer value. This nonce is read by the receiving device and copied into the nonce field of the corresponding response. When a device that has previously sent a request, gets this response, it can associate it to the formerly sent request. Until the response arrives, the requesting device saves the request and keeps its field contents. This way, no fields need to be redundantly repeated by the response, except for the nonce. The response acts at least as an acknowledgment from the device for having received the request.

Each message has one field to identify the sender user and one field to identify the sender device respectively. With these fields, the receiver can first identify the sending device by its *DID*. If the sending device's *DID* is unknown to the receiver (e.g. because the sender user has a new device), it still can associate the message to the correct contact by the *UID* field.

Additionally, each message has one field for the receiver device id which is filled by the sender. If a device receives a message that does not match its own *DID*, it discards the message, since it is not the intended receiver. This can happen e.g. due to changing IP addresses. In this case, no response is sent to the sender. This way, the sender device notices that the device it wanted to reach is not available.

Lastly, each device communicates its own capability in form of an abstract Communication Costs Value (CCV). The interpretation of this value is left to resource-aware Decision Engines.

Table 5.4 displays all general fields that are set in every message, whether request or response.

Table 5.5 shows an overview on all message types in SocioPath, including the section in this chapter with detailed explanations.

Table 5.4: *General fields inside every SocioPath request and response message*

Data type	Field name	Description
`8 bit Unsigned Integer`	messageType	Message Type ID (see Table 5.5)
`UID`	senderUserId	*UID* of the user who owns the device that sends the message
`DID`	senderDeviceId	*DID* of the device that sends the message
`DID`	receiverDeviceId	*DID* of the device that receives the message
`Integer`	msgNonce	Nonce for identifying according response to a request
`Integer`	senderDeviceCCV	Sending device's Communication Costs Value (CCV) for usage by resource-aware Decision Engines
`Integer`	stateNonce	Required to distinguish new notifications from re-sends of old ones during a state repair. State repairs are discussed in Chapter 6.

5.4.2 Messages for Data Object Delivery

This messages presented in this section offer a basic form of data object transport without decoupling notifications and data objects yet. Decoupling was already discussed for SODESSON

Table 5.5: *Overview on message type IDs*

Message Type ID	Name	Section
1	NOTIFY$_{\text{Req}}$	5.4.2 / 5.7.2
2	NOTIFY$_{\text{Rsp}}$	5.4.2 /5.7.2
11	RETRIEVE$_{\text{Req}}$	5.7.3
12	RETRIEVE$_{\text{Rsp}}$	5.7.3
21	NEWCONTACT$_{\text{Req}}$	5.6.3
22	NEWCONTACT$_{\text{Rsp}}$	5.6.3
31	STATE$_{\text{Req}}$	6.3
32	STATE$_{\text{Rsp}}$	6.3

(Section 4.3.6) and will be later discussed for SocioPath (Section 5.7). The basic form of delivery which is described here gets extended there.

NOTIFY$_{\text{Req}}$ **and** NOTIFY$_{\text{Rsp}}$ **messages types**: If decoupling is not used, these message types are used for data object forwarding and delivery, i.e. the data content is piggybacked in the message. When decoupling, these message types are used for notifications about a published data object without the content.

In SocioPath, data objects can be transported via NOTIFY$_{\text{Req}}$ messages. For the sake of simplicity, the general concept of delivering data objects is now explained by NOTIFY$_{\text{Req}}$ messages only, i.e. it is assumed that all data objects are piggybacked. Additionally, only basic fields of a NOTIFY$_{\text{Req}}$ message are explained for now.

The fields of a NOTIFY$_{\text{Req}}$ for a data object δ_t and corresponding NOTIFY$_{\text{Rsp}}$ message (in addition to the general message fields in Table 5.4) are displayed in Tables 5.6 and 5.7.

The NOTIFY$_{\text{Rsp}}$ contains a general purpose "ok" boolean flag which can be used differently e.g. depending on the topic. As an example, the maintenance topic *SubscribeMe* makes use of the flag (see Section 5.3.2).

Table 5.6: *Basic fields inside a* NOTIFY$_{Req}$ *message*

Data type	Field name	Description
`String`	topicTitle	Topic title of t ($title_t$)
`UID`	topicOwner	UID of the topic owner of t (UID_{TO_t})
`Integer`	objId	Object ID of δ_t (OID_{δ_t})
`Integer`	publishTime	Timestamp when data object was published. Used together with *objId* to uniquely identify the data object (see Section 4.3.1) and recognize obsolete notifications (see Section 4.3.7).
`BinaryData`	dataContent	Holds the data object's content.
`Integer`	ttl	Time-to-live for δ_t in seconds since the publish timestamp, see Section 4.3.5 about the Local Data Storage for details.
`UID`	publisherUserId	User ID of the data object's publisher. Used to identify the original publisher, since this field can differ from *senderUserId*.
`Boolean`	forward	This flag indicates to the Decision Engine whether the receiving device is asked to forward the NOTIFY$_{\text{Req}}$ message to other devices.

Table 5.7: *Fields inside a* NOTIFY$_{Rsp}$ *message*

Data type	Field name	Description
`Boolean`	ok	General purpose for ACK / NACK

5.4.3 Maintenance Topics

A maintenance data structure is updated via NOTIFY$_{\text{Req}}$ messages with the according data in the *dataContent* field. Table 5.8 gives an overview on the different maintenance topics.

The case for updating the Own Topics List as well as the Subscription Lists is discussed in this section. Updating Contact List and Devices List will be discussed separately in Section 5.6.

Creating a New Topic

Section 4.3.3.2 described how the SODESSON middleware handles the creation of a new topic. Specifically, the definition of allowed publishers and allowed subscribers had to be sent from the topic owner's device to a provider device which is responsible for access rights control. They hold the information about allowed publishers, allowed subscribers and subscribers.

In SocioPath, these devices are the topic owner's devices themselves. This is the result that in User-Centric Networking only the topic owner's devices hold the information about allowed publishers, allowed subscribers and subscribers and therefore must be responsible for access control.

Therefore, the topic owner device a_1 where the topic owner A has created and defined the sets of allowed publishers and allowed subscribers must publish this information as a data object with

Table 5.8: *Maintenance Topics*

Topic Title	Allowed Publishers	Subscribers	Description
OwnTopics	Topic Owner	Topic Owner	For updates in the topic owner's Contact List. See Sections 5.5.1 and 5.5.2) for details.
OwnSubscriptions	Topic Owner	Topic Owner	For updates in the topic owner's Subscriptions List. See Section 5.5.2 for details
SubscribeMe	Topic Owner Contacts	Topic Owner	For subscribing to a topic: Contacts publish to this topic, topic owner as the only subscriber gets notified. See Section 5.5.2 for details for details
UnsubscribeMe	Topic Owner Contacts	Topic Owner	For unsubscribing from a topic: Contacts publish to this topic, topic owner as the only subscriber gets notified. See Section 5.5.2 for details for details
OwnContacts	Topic Owner	Topic Owner	For updates in the topic owner's Contact List. See Section 5.6.3 for details.
Devices	Topic Owner	Topic Owner Contacts	For changes in the topic owner's device pool, updates the topic owner's and his contacts' Devices Lists. See Section 5.6.2 for details.

the maintenance topic {"OwnTopics", A}. As discussed in Section 5.3.2, A is the only subscriber and allowed publisher for this maintenance topic.

The content of the data object is a serialization of the new topic's title, the set of allowed publishers, the set of allowed subscribers and the optional privacy level.

The objId of the data object is a hash over the new topic's title. If the access rights of the new topic need to be updated later (e.g. a new allowed subscriber shall be added), then an update, i.e. a new data object with the same objId and a newer timestamp must be published. By using the hash, the objId can be easily reproduced, as long as the topic title stays the same. The objId does not have to be stored in a separate data structure.

Since A is a subscriber, this data object must be delivered to all devices of A. If a_1 is the only device of A, nothing else has to be done and the topic creation is finished.

Subscribing

Subscribers in SocioPath are maintained in the Own Topics List of the respective topic owner's devices. If a contact B of the topic owner A wants to subscribe to A's topic t, B needs to publish a data object with A's maintenance topic {"SubscribeMe", A}. If B is an allowed subscriber, A's topic owner devices enter B as a subscriber into their Own Topics List and the requesting device gets a positive (otherwise negative) response.

The content of the data object is the topic title that B requests to subscribe to.

The objId of the data object is a hash over the new topic's title and the requesting subscriber (here: B). By hashing this combination, subscriptions of two differents users for the same topic yield different objIds. Likewise, subscriptions of the same user for two different topics yield different objIds. Note that all subscription requests for topics where A is topic owner result in data objects for the topic {"SubscribeMe", A}, therefore it has to be ensured that two parallel subscriptions do not overwrite each other. Hashing over topic and user is a simple way to achieve this.

Since A is subscriber of {"SubscribeMe", A}, this data object must be delivered to all devices of A.

For each NOTIFY$_{\text{Req}}$ message the requesting device b sends to a topic owner device a, b receives a NOTIFY$_{\text{Rsp}}$ message from a. If B belongs to the allowed subscribers according to a's Own Topics List, the *ok* flag in the NOTIFY$_{\text{Rsp}}$ message is set to true, otherwise to false. If b receives at least one true *ok*, b enters the subscribed topic into its Subscriptions List.

Finally, if b at least one true *ok*, it is passed to the App Manager, which notifies the subscribed application about the successful subscription.

A special case can occur where A set B as an allowed subscriber (e.g. on a_1), but the information has not propagated to a_2 yet. In this case, the subscription request would be falsely denied by a_2. Here, a_1 and a_2 need to synchronize first by state repairs (see Chapter 6). It is up to the application to retry subscription requests.

After successfully subscribing, the Subscriptions Lists of user B (the subscriber) needs to be updated on all of his devices. This update is done by a data object with the maintenance topic {"OwnSubscriptions", B}. Here, B is the topic owner and the only subscriber and allowed publisher. The NOTIFY$_{\text{Req}}$ message has the following field values.

The content of the data object is a serialization of the formerly subscribed topic title and its topic owner's UID. The objId of the data object is a hash of said serialization. Note that all Subscription List updates results in data objects for the topic {"OwnSubscriptions", B}, therefore it has to be ensured that two different entries for the Subscriptions List do not overwrite each other. This is achieved by the hash.

After each device of B has received the NOTIFY$_{\text{Req}}$ message, it updates its Subscriptions List accordingly.

Unsubscribing

Unsubscribing works analogously to subscribing. The main difference is the topic being used by b_1 for the unsubscription, which is "UnsubscribeMe". Also, a topic owner device always responds with a NOTIFY$_{\text{Rsp}}$ message with *ok* set to true, regardless whether B is actually subscribed to the topic in question or whether the topic even exists. The rest of the process is similar to subscription: the topic owner devices have to update their Own Topics Lists and the devices of the unsubscribing user have to update their Subscriptions Lists.

5.4.4 CCV and Forward Flag

This section discusses the two properties CCV and forward flag, that each SocioPath message offers to Decision Engines. These properties can be used to reduce NOTIFY$_{\text{Req}}$ and NOTIFY$_{\text{Rsp}}$ strain on a single device.

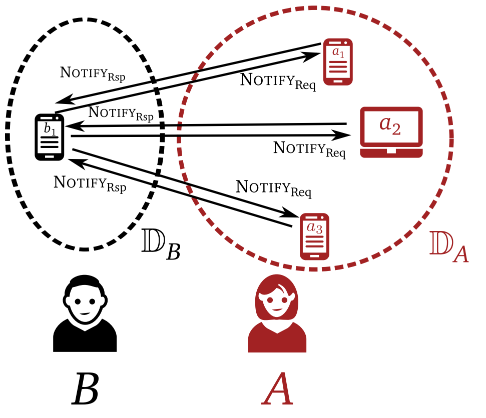
Figure 5.4 shows a situation with topic owner A and an allowed publisher B that publishes a data object on his device b_1. The three devices of topic owner A must receive that data object.

Two possibilities are shown: in Figure 5.4a, b_1 sends the NOTIFY$_{\text{Req}}$ messages to all three topic owner devices by itself and receives in turn three NOTIFY$_{\text{Rsp}}$ messages. In Figure 5.4b, b_1 sends the NOTIFY$_{\text{Req}}$ message to a_2 only, while a_2 sends the NOTIFY$_{\text{Req}}$ message to the remaining devices of A. Here, b_1 receives only one NOTIFY$_{\text{Rsp}}$ message.

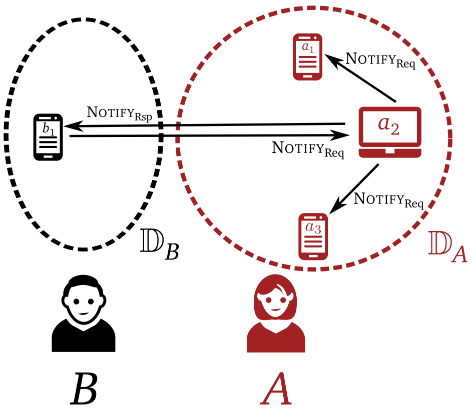
In the second case, b_1 only has to deal with two messages instead of six in total. Instead, the majority of the work is offloaded to a_2 which has earlier announced its CCV to all devices it knows, including b_1. Due to a_2's low CCV, b_1 has chosen a_2 as the only target device and set the forward flag in the NOTIFY$_{\text{Req}}$ message to ask a_2 to deliver the message to the other devices of A.

The same issue can also arise if there is only one user involved. Assume that topic owner A uses her device a_1 to create a new topic. If there are other topic owner devices (a_2, a_3, ...), the maintenance data object must be delivered to all these devices of A. a_1 selects *at least one* other device from its Device List and sends a NOTIFY$_{\text{Req}}$ message holding the data object to this device selection. This selection is trivial if there is only one other device a_2.

However, if there are three or more topic owner devices, there are multiple possibilities. Figure 5.5 shows two possibilities in an example: in Figure 5.5a, a_1 sends the NOTIFY$_{\text{Req}}$ messages all by itself to $\{a_2, \ldots, a_5\}$. In Figure 5.5b, a_1 offloads the NOTIFY$_{\text{Req}}$ message to a_2, with the *forward* flag set to true. This makes a_2 send the NOTIFY$_{\text{Req}}$ message to the remaining devices. A reason for this offloading might be the communication costs according to the device's CCV. If a_1 has higher communication costs than a_2, it is cheaper for A to let a_1 send only one NOTIFY$_{\text{Req}}$ message instead of four NOTIFY$_{\text{Req}}$ messages and leave the rest to a_2.



(a) Device a_1 sends NOTIFY$_{\text{Req}}$ messages to all other devices by itself



(b) Device a_1 offloads NOTIFY$_{\text{Req}}$ message to a_2 first, a_2 sends NOTIFY$_{\text{Req}}$ message to the remaining devices

Figure 5.4: *Two possibilities for the publishing device to notify the topic owner devices*

Clearly, resource-awareness is required for solving this problem, therefore the task of interpreting CCVs and reacting accordingly is left to the Decision Engine. Solutions to this problem are discussed in Chapter 7.

5.5 Example Workflow of Data Object Delivery

In Section 4.3.3, the publishing of a data object was described with the help of an example. That example contained four steps:

(1) Binding an application to SODESSON

(2) Creating a new topic

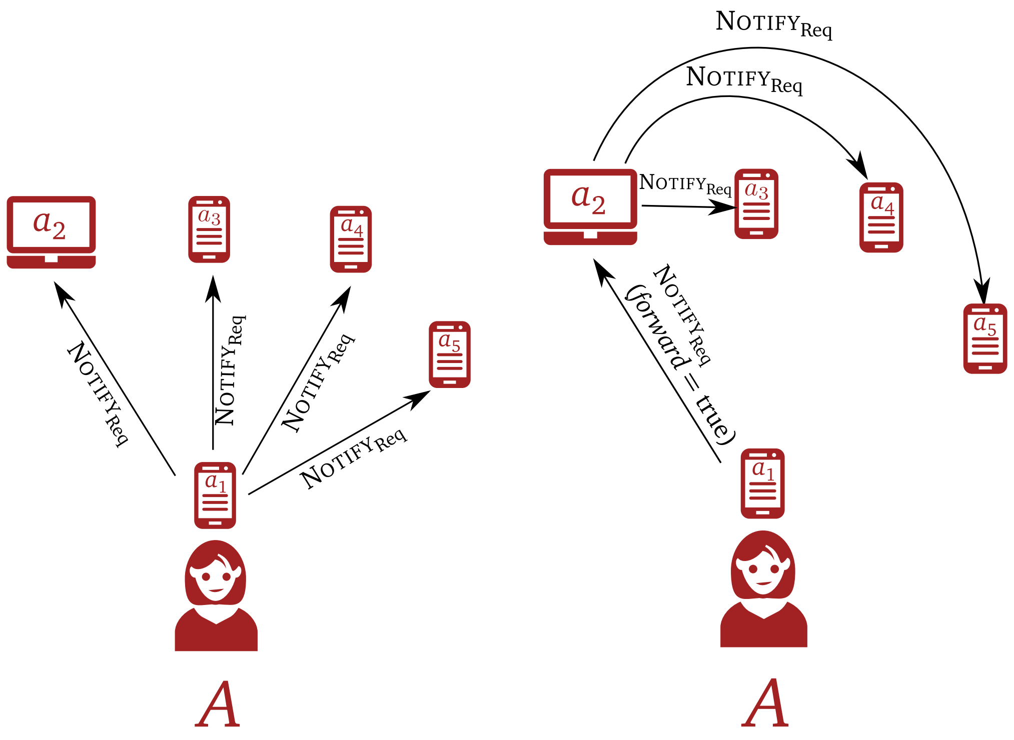
(3) Subscription

(4) Publishing a new data object

The first step can be ignored in the scope of a DDP, since that step is solely executed between application and middleware on the same device and does not involve inter-device communication. The steps for topic creation, subscription and publishing / delivering a data object shall here be discussed for SocioPath. The steps will be explained with the help of the blog post example that was already used for explaining data object delivery between applications and the SODESSON middleware on the same device.

5.5.1 Creating a new topic

Section 4.3.3.2 described how the SODESSON middleware handles the creation of a new topic. Here, user A created a new topic "App1_Blog" and defined A as an allowed publisher and $\{A, B, C\}$



(a) Device a_1 sends NOTIFY$_{\text{Req}}$ messages to all other topic owner devices by itself

(b) Device a_1 offloads NOTIFY$_{\text{Req}}$ message to a_2 first, a_2 sends NOTIFY$_{\text{Req}}$ message to the remaining topic owner devices

Figure 5.5: *Creating a new topic: Two possibilities for updating the topic owner devices*

as allowed subscribers. Figure 5.6 is an extended version of Figure 4.3. It depicts this procedure again and adds how SocioPath handles the steps ③ to ⑤ with the DDP's inter-device communication, which was not discussed earlier.

Topic owner A uses a_1 to create the topic "App1_Blog". After the application has called SODESSON's updateTopic method, an entry in the Own Topics list on a_1 is created with the values shown in Table 5.9.

Step ③: Since the topic creation information is a data object with the maintenance topic {"OwnTopics", A"}, the data object gets stored in the Local Data Storage.

Step ④: Device a_1 looks up a_2 in a_1's Device List and sends a NOTIFY$_{\text{Req}}$ message to a_2[1]. The NOTIFY$_{\text{Req}}$ message has the fields set as shown in Table 5.10.

Step ⑤: On arrival of the NOTIFY$_{\text{Req}}$ message, SocioPath on a_2 recognizes the maintenance topic "OwnTopics". It deserializes the *dataContent* field, and creates an entry in its Own Topics list with the same value as a_1 did, as shown in Table 5.9. Additionally, the data object gets stored in the Local Data Storage of a_2.

[1]For the sake of simplicity, only two devices a_1 and a_2 are assumed. Any additional topic owner devices would receive analogous NOTIFY$_{\text{Req}}$ messages.

Table 5.9: *Creating a new topic: Own Topics List entry for "App1_Blog" (blog example)*

Data type	Field name	Description
`String`	topicTitle	"App1_Blog"
`Array<UID>`	allowedPubs	$\{UID_A\}$
`Array<UID>`	allowedSubs	$\{UID_A, UID_B, UID_C\}$
`Array<UID>`	subscribers	{}
`Enum`	privacyLevel	3
`Integer`	lastUpdate	142571829

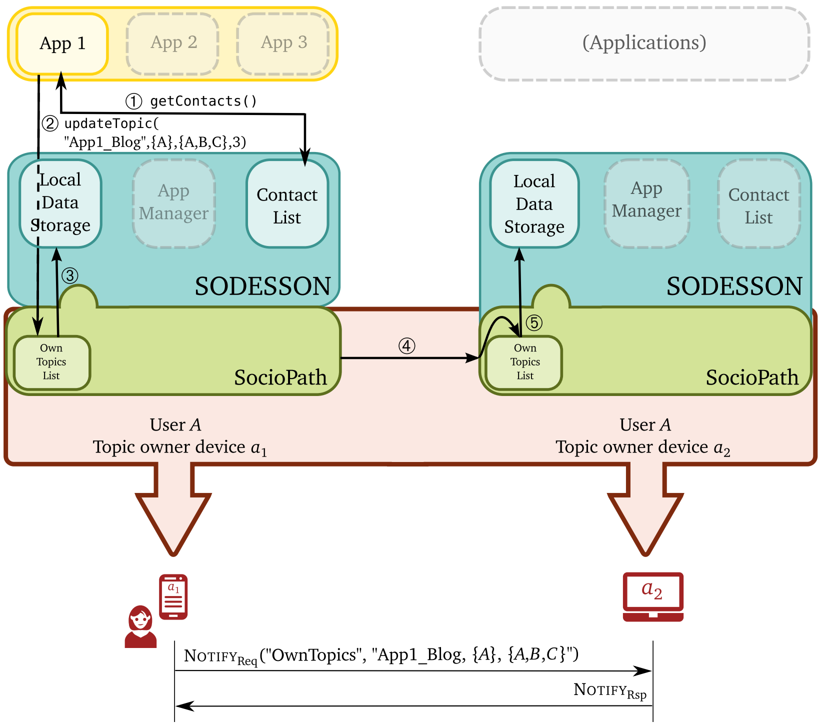


Figure 5.6: *Creating a new topic: a_1 updates the other topic owner device a_2 (blog example)*

5.5.2 Subscription

Section 4.3.3.3 described how another user subscribed to A's topic "App1_Blog" and how this was handled by the SODESSON middleware. Figure 5.7 depicts this procedure and adds how SocioPath handles the sub-steps ③ to ⑦ on the DDP layer. In Section 4.3.3.3 the DDP was abstracted and not discussed.

Step ③: Since the topic creation information is a data object with the maintenance topic {"SubscribeMe", A"}, the data object gets stored in the Local Data Storage of b_1.

Step ④: Device b_1 looks up a_1 and a_2 in b_1's Device List and sends a NOTIFY$_{Req}$ message to each of these. The NOTIFY$_{Req}$ message for a_1 has the fields set as shown in Table 5.11, the NOTIFY$_{Req}$ message for a_2 has analogous values.

Table 5.10: *Fields of the* NOTIFY$_{Req}$ *message on topic creation (blog example)*

Data type	Field name	Value
`8 bit Unsigned Integer`	messageType	1
`UID`	senderUserId	UID_A
`DID`	senderDeviceId	DID_{a_1}
`DID`	receiverDeviceId	DID_{a_2}
`Integer`	msgNonce	789
`String`	topicTitle	"OwnTopics"
`UID`	topicOwner	UID_A
`Integer`	objId	hash("App1_Blog")
`Integer`	publishTime	142571829
`BinaryData`	dataContent	"App1_Blog, $\{UID_A\}$, $\{UID_A, UID_B, UID_C\}$, 3"
`Integer`	ttl	0
`UID`	publisherUserId	UID_A
`Boolean`	forward	false

Step ⑤: The topic owner devices, i.e. A's devices a_1 and a_2, now infer the following pieces of information: B (*publisherUserId*) wants to subscribe (*topicTitle*) to a topic that A (*topicOwnerUserId*) owns. The topic is called "App1_Blog" (*dataContent*). The topic owner device now checks in its Own Topics List whether A is an allowed subscriber for "App1_Blog".

Step ⑥: Both a_1 and a_2 infer from their Own Topic List that B is an allowed subscriber. They send a NOTIFY$_{Rsp}$ message with ok = true to b_1. a_1 now enters "{App1_Blog, B}" into its Subscriptions List.

Step ⑦: Since b_1 has received at least one NOTIFY$_{Rsp}$ message with ok = true, a success indication is sent to the App Manager which in turn notifies App 1 about the subscription success ⑧.

Now, b_1 notifies B's other devices (here: b_2) about the successful subscription, so they update their Subscriptions List. The respective NOTIFY$_{Req}$ message is shown in Table 5.12.

5.5.3 Publishing a new data object

After the previous maintenance, finally the blog post is published. Similarly to the previous steps, that data object is sent via a NOTIFY$_{Req}$ message to the subscribers.

Assume that user B is not only subscriber of the topic "App1_Blog @ A", but also an allowed publisher. Non-topic owner, but allowed publisher, B publishes a new data object via his device b_1. Since only the topic owner A knows the subscribers, the data object has to be sent to the topic owner devices first. Table 5.13 shows an example for such a NOTIFY$_{Req}$ message.

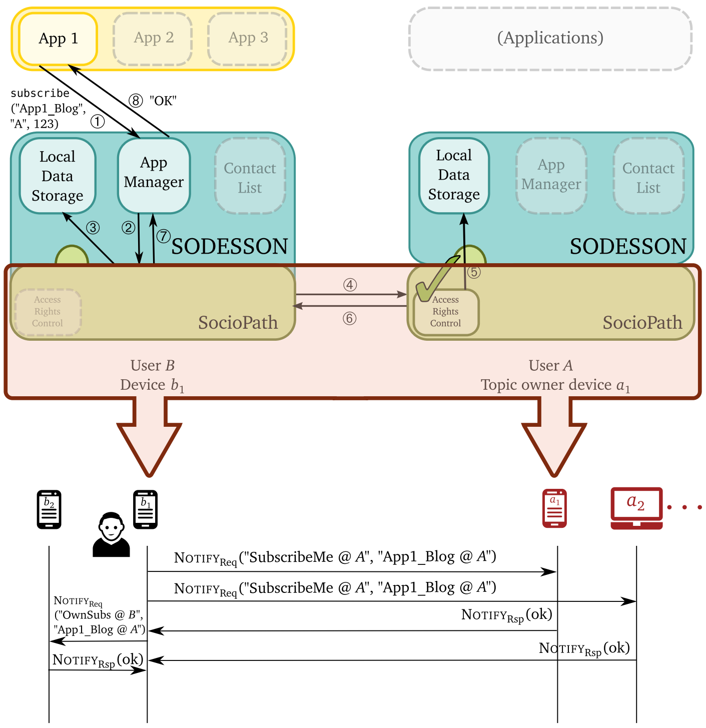


Figure 5.7: *Subscribing: b_1 sends a subscription request for "App1_Blog @ A" to the topic owner devices, i.e. all of A's devices*

5.6 Additional Maintenance

This section covers additional maintenance tasks that have not been discussed so far: adding new devices and adding new contacts. These also require publishing data objects with maintenance topics.

5.6.1 Initial Setup

For the initial setup on the very first device a_1 in $\mathbb{D}_A$ in SODESSON, i.e. Contact List and App Manager, see Section 4.4.2. For SocioPath, a_1 is the only entry in the Devices List. The Own Topics List contains only the maintenance topics with full access rights given to A as the topic owner, the Subscriptions List is filled accordingly. The Local Data Storage is empty.

5.6.2 Adding new devices

If user A wants add a new device and wants to add it into his device pool $\mathbb{D}_A$, he has to call the importUser method in SODESSON and import his public/private keypair first. Afterwards, all of his other devices and those of his contacts must learn that there is a new device. This way, NOTIFY$_{Req}$ messages can be sent to the new device as well. The new device a_j has be to paired with an existing device a_i of A. Pairing requires that a_i and a_j are available for each other,

Table 5.11: *Fields of the* NOTIFY$_{Req}$ *message on subscribing (blog example)*

Data type	Field name	Value
`8 bit Unsigned Integer`	messageType	1
`UID`	senderUserId	UID_B
`DID`	senderDeviceId	DID_{b_1}
`DID`	receiverDeviceId	DID_{a_1}
`Integer`	msgNonce	5678
`String`	topicTitle	"SubscribeMe"
`UID`	topicOwner	UID_A
`Integer`	objId	hash("App1_Blog", UID_B)
`Integer`	publishTime	142573456
`BinaryData`	dataContent	"App1_Blog"
`Integer`	ttl	0
`UID`	publisherUserId	UID_B
`Boolean`	forward	false

can locate each other and exchange data. During this pairing process, the following types of information gets copied from a_i to a_j:

- Contact List
- Devices List
- Own Topics List
- Subscriptions List

The exchange is performed with BOOTSTRAP$_{\text{Req}}$ and BOOTSTRAP$_{\text{Rsp}}$ messages. The BOOTSTRAP$_{\text{Req}}$ has no additional fields besides the general SocioPath message fields (see Section 5.4). The BOOTSTRAP$_{\text{Rsp}}$ message holds arrays of serialized entries of the Contact List, Devices List, Own Topics List and Subscriptions List respectively, as shown in Table 5.14.

Data objects in the Local Data Storage of a_i are not part of the BOOTSTRAP$_{\text{Rsp}}$ message. Instead, they get transferred to a_j via the regular STATE repair mechanism, as will be described in Chapter 6.

After receiving the BOOTSTRAP$_{\text{Rsp}}$ message, a_j introduces itself to any other devices of *A* and all of *A*'s contacts. a_j publishes its own Device List entry as a data object with maintenance topic {Devices, *A*}. Both *A* and all of *A*'s contacts are subscribers to this topic, hence all of user *A*'s devices and all *A*'s contacts' devices receive the data object and update their Devices List.

5.6.3 Adding contacts

After two contacts have successfully added to each other's SODESSON Contact List (see Section 4.4.3), their devices need to exchange information in SocioPath as well. One device a_i of user

Table 5.12: *Fields of the* NOTIFY$_{Req}$ *message for updating OwnSubscriptions list between* b_1 *and* b_2 *(blog example)*

Data type	Field name	Value
`8 bit Unsigned Integer`	messageType	1
`UID`	senderUserId	UID_B
`DID`	senderDeviceId	DID_{b_1}
`DID`	receiverDeviceId	DID_{b_2}
`Integer`	msgNonce	9876
`String`	topicTitle	"OwnSubscriptions"
`UID`	topicOwner	UID_B
`Integer`	objId	hash("App1_Blog", UID_A)
`Integer`	publishTime	142574963
`BinaryData`	dataContent	"App1_Blog, UID_A"
`Integer`	ttl	0
`UID`	publisherUserId	UID_B
`Boolean`	forward	false

A has to pair with a device b_j of user B. Pairing requires that a_i and b_j are available for each other, can locate each other and exchange data. During this pairing process, each device sends information about other devices of the user:

- $a_i \rightarrow b_j$: Devices of A ($\mathbb{D}_A$)
- $b_j \rightarrow a_i$: Devices of B ($\mathbb{D}_B$)

The exchange is performed with NEWCONTACT$_{\text{Req}}$ and NEWCONTACT$_{\text{Rsp}}$ messages. Since both messages transport the same type of information, their fields are identical. Their fields are described in Table 5.15.

After the pairing, a_i enters the devices in $\mathbb{D}_B$ into its Devices List. Subsequently, a_i publishes this information (new contact B information and his devices) as a data object to the maintenance topic {"OwnContacts", A}.

The content of the data object is a serialization of B's UID, B's alias and B's public key, as well as serializations of the respective Devices Lists entries.

The objId of the data object is a hash of the new contact's UID_B. By using this hash as the objId, two different new contacts yield two different objIds. Note that all newly added contacts result in data objects for the topic {"OwnContacts", A}, therefore it has to be ensured that two parallel subscriptions do not overwrite each other.

Device b_j proceeds in an analogous way. Figure 5.8 sketches an example protocol flow.

Table 5.13: *Fields of the* NOTIFY$_{Req}$ *message on topic creation (blog example)*

Data type	Field name	Value
`8 bit Unsigned Integer`	messageType	1
`UID`	senderUserId	UID_B
`DID`	senderDeviceId	DID_{b_1}
`DID`	receiverDeviceId	DID_{a_1}
`Integer`	msgNonce	1243
`String`	topicTitle	"App1_Blog"
`UID`	topicOwner	UID_A
`Integer`	objId	42
`Integer`	publishTime	143656251
`BinaryData`	dataContent	"<title>We have a new dog!</ti..."
`Integer`	ttl	0
`UID`	publisherUserId	UID_B
`Boolean`	forward	false

5.7 Decoupling notifications and data object retrievals

In Section 4.3.6, it was described how SODESSON supports the decoupling of a data object's metadata and its actual content. By doing so, a resource-aware device can in a first step learn about the existence of a new object (indicated by a small *notification* including topic, size, etc.) and then decide on its own whether to *retrieve* the data object's content. The reason for this decoupling is resource awareness. For example, a user would not want to download a video file on his smartphone if it is restricted in storage space and has a volume-based mobile connection, especially if he never consumes videos on his smartphone.

SocioPath supports this decoupling. Published data objects which exceed a certain size are at first indicated by a notification which is pushed to each subscriber device. The respective size

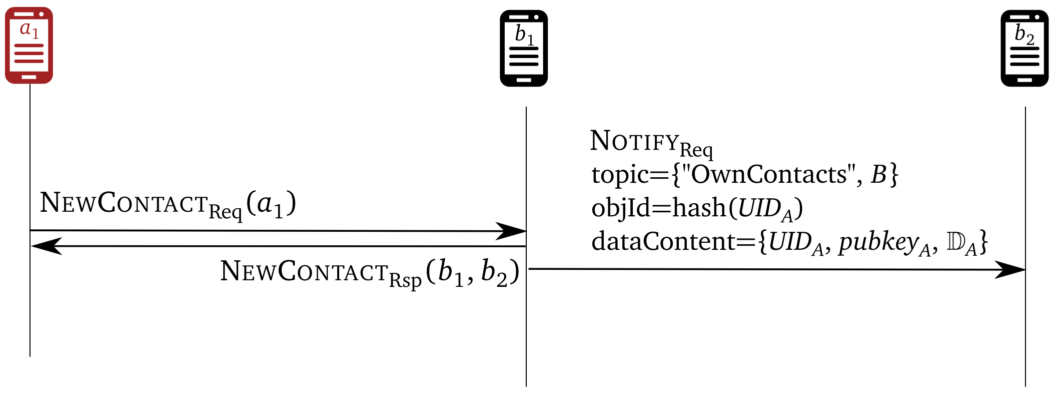

a_1
b_1
b_2
NEWCONTACT$_{\text{Req}}(a_1)$
NEWCONTACT$_{\text{Rsp}}(b_1, b_2)$
NOTIFY$_{\text{Req}}$
topic={"OwnContacts", B}
objId=hash(UID_A)
dataContent={UID_A, $pubkey_A$, $\mathbb{D}_A$}

Figure 5.8: *Adding contact procedure*

Table 5.14: *Fields inside a* BOOTSTRAP$_{Rsp}$ *message. Message is sent from device* a_i *by user A.*

Data type	Field name	Description
`8 bit Unsigned Integer`	messageType	Message Type ID (see Table 5.5)
`Array<ContactListEntry>`	contacts	Full Contact List as stored on a_i, including their UIDs, public keys and aliases
`Array<DeviceListEntry>`	devices	Full Devices List as stored on a_i, with A's and A's contacts devices. Each entry includes the device's owner's UID, the device's DID and the device's locator
`Array<OwnTopicsEntry>`	ownTopics	Full Own Topics List of A as stored on a_i. Each entry includes the topic title, allowed publishers, allowed subscribers and subscribers.
`Array<Topic>`	subscribedTopics	Full Subscribed Topics List of A as stored on a_i. Each entry includes the topic title, allowed publishers, allowed subscribers and subscribers.

Table 5.15: *Fields inside a* NEWCONTACT$_{Req}$/ NEWCONTACT$_{Rsp}$ *message. Message is sent from device* a_i *by user A.*

Data type	Field name	Description
`Array<DeviceListEntry>`	devices	Devices List entries of the user of the sending device. Each entry includes the device's owner's UID, the device's DID and the device's locator

limit is to be configured in SocioPath's **Decision Engine (DE)**. Based on the metadata in this notification, either the user / application or the DE can decide whether to *retrieve* the actual data object.

The decoupling in SocioPath is established by four different message types: NOTIFY$_{\text{Req}}$ / NOTIFY$_{\text{Rsp}}$ and RETRIEVE$_{\text{Req}}$ / RETRIEVE$_{\text{Rsp}}$. For small data objects the overhead of an additional RETRIEVE message exchange might be higher than the size of the data object itself. In such a case, the data content can be **piggybacked** inside the *dataContent* field of the NOTIFY$_{\text{Req}}$. This is exactly what happened when the steps of publishing a data object were revisited in Section 5.5. Again, it is up to the Decision Engine on the device sending the NOTIFY$_{\text{Req}}$ message to decide whether to piggyback the data content.

In Section 5.3, it was established that each published data object δ_t shall be delivered to the full device pool $\mathbb{D}_{\delta_t}$.

Decoupling relaxes this requirement: only *notifications* shall be delivered to these devices. If the data object content is not piggybacked, each device can autonomously decide whether to retrieve the content.

The decoupled process for delivering data objects will be discussed with an introductory example (Section) and afterwards for notifications (Section 5.7.2) and retrievals (Section 5.7.3) respectively.

One challenge lies in the selection of source devices that should be disclosed to subscribers by the topic owner. This selection has to comply with the demanded privacy level. This problem is discussed in Section 5.7.4.

5.7.1 Decoupling example

In this section, an introductory example for decoupling is shown. The example shows how devices can act differently in the same situation, based on their DE's behaviour. In the following, all names in **blue** are Decision Engine events. This indicates a situation where the Decision Engine has to decide how its device should proceed. Decision Engine events are discussed in detail in Chapter 7.

Figure 5.9 outlines two typical sequences of NOTIFY$_{\text{Req}}$ / NOTIFY$_{\text{Rsp}}$ and RETRIEVE$_{\text{Req}}$ / RETRIEVE$_{\text{Rsp}}$, respectively. Topic owner and allowed publisher A has published a new data object on device a_1. B and C are subscribers. The Decision Engine gets informed about this **PublishFromApp** event and enforces the sending of NOTIFY$_{\text{Req}}$ messages to $\mathbb{D}_A = \{(a_1), a_2\}$, $\mathbb{D}_B = \{b_1\}$ and $\mathbb{D}_C = \{c_1\}$. Upon arrival, each device instantly sends a NOTIFY$_{\text{Rsp}}$ message as an acknowledgment.

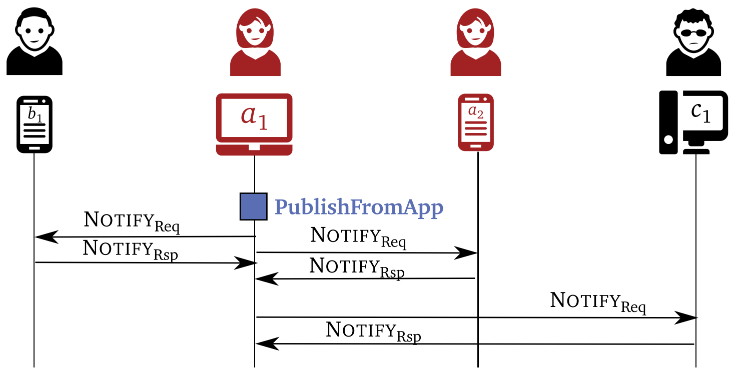
In Figure 5.9 (a) the data object content is piggybacked in the NOTIFY$_{\text{Req}}$ messages and no further action is needed on a_1 or a receiving device (a_2, b_1, c_1). This is analogous to the previous discussion of publishing a data object in Section 4.3.3.4.

In Figure 5.9 (b) the data content is not piggybacked and the NOTIFY$_{\text{Req}}$ indicates that the notified devices have to retrieve the data content. The Decision Engine on device c_1 is informed about this by the **RetrieveRequired** event and deems it acceptable to instantly retrieve the data content and enforces sending a RETRIEVE$_{\text{Req}}$ to a_1. The Decision Engine on a_1 in turn decides to accepts this request and denotes this with a positive RETRIEVE$_{\text{Rsp}}$.

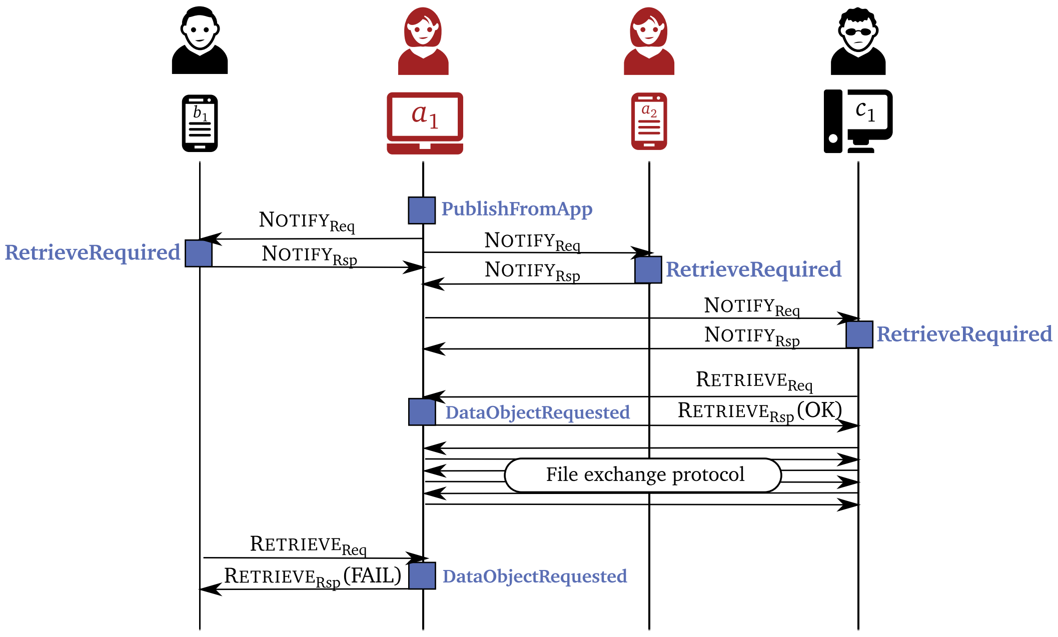
Due the positive RETRIEVE$_{\text{Rsp}}$, a_1 initiates the file transfer with c_1. The actual data transfer is not performed by the SocioPath protocol, but instead by a state-of-the-art file exchange protocol such as PPSPP , a peer-to-peer streaming protocol which can also be used for static files. The RETRIEVE$_{\text{Req}}$/ RETRIEVE$_{\text{Rsp}}$ is thus only a resource-aware negotation whether file exchange is currently acceptable.

Some time later, the Decision Engine on b_1 decides to enforce sending RETRIEVE$_{\text{Req}}$ to a_1. A possible reason for this could be the change from a mobile connection to free WiFi. a_1 however declines via a RETRIEVE$_{\text{Rsp}}$ (e.g. because it exceeded its data volume by providing the data content to c_1) and no data object content is delivered to b_1 at that time. This can be dealt with in several ways: The Decision Engine on b_1 can periodically send a RETRIEVE$_{\text{Req}}$ to a_1, but if a_1 declines every time, b_1 will never retrieve the data content. Instead, a_1 could notify b_1 that c_1 is another potential source.

The example shows that large parts of the protocol behaviour are tied to the DE. Hence, the design of the Decision Engine has a strong impact on how SocioPath behaves and performs in certain situations. The main protocol invariants of SocioPath are the different message types as well as the specification of fix events every Decision Engine has to react to.



(a) Example sequence diagram for NOTIFY message flows



(b) Example sequence diagram for NOTIFY and RETRIEVE message flows

Figure 5.9: *Sequence diagram for* NOTIFY *and* RETRIEVE. *Decision Engine events are denoted in blue.*

5.7.2 Notifications

In Section 5.4.2 and Table 5.6, a basic version of the NOTIFY$_{\text{Req}}$ message was introduced. In that basic version all data content was piggybacked inside the NOTIFY$_{\text{Req}}$ message itself.

For notifications decoupled from the data object content, these have to be extended with additional fields.

Extended notification messages

The extended version of a NOTIFY$_{\text{Req}}$ message (in addition to the general message fields in Table 5.4) is displayed in Table 5.16. The fields printed in **bold** are new compared to the basic version. The field *dataContent* is not new, but now has one of two different functions, depending on whether the *retrieveRequired* boolean flag is set to true or false.

Table 5.16: *Additional fields inside a* NOTIFY$_{Req}$ *message.*

Data type	Field name	Description
`String`	topicTitle	Topic title of t ($title_t$)
`UID`	topicOwner	UID of the topic owner of t (UID_{TO_t})
`Integer`	objId	Object ID of δ_t (OID_{δ_t})
`Integer`	publishTime	Timestamp when data object was published. Used together with *objId* to uniquely identify the data object (see Section 4.3.1) and recognize obsolete notifications (see Section 4.3.7).
`Integer`	**contentSize**	**Data object content size in bytes. This information helps the Decision Engine of b_1 to decide if and when to retrieve the data object – unless it is already piggybacked by the sender.**
`Boolean`	**retrieveRequired**	**Flag that indicates whether the data object must be retrieved (true) or was piggybacked (false).**
`BinaryData`	**dataContent**	***If retrieveRequired = false*: holds the data object's content (piggybacked). *If retrieveRequired = true*: holds additional information about the data object for the file exchange protocol.**
`Integer`	ttl	Time-to-live for δ_t in seconds since the publish timestamp, see Section 4.3.5 about the Local Data Storage for details.
`UID`	publisherUserId	User ID of the data object's publisher. Used to identify the original publisher, since this field can differ from *senderUserId*.
`Boolean`	forward	This flag indicates to the Decision Engine whether the receiving device is asked to forward the NOTIFY$_{\text{Req}}$ message to other devices.
`Array<DID>`	**sources**	**List of devices that are known to the device which sends the NOTIFY$_{\text{Req}}$ message to hold the data object's content. At least the device on which the data object was originally published, is entered here.**

Sending and handling notifications

Each device that sends a NOTIFY$_{\text{Req}}$ message and holds the data object's content can decide whether to piggyback the data object. This decision is the task of the sending device's DE. The fields *retrieveRequired* and *dataContent* are set accordingly:

- If piggybacked: *retrieveRequired* = false and *dataContent* holds the data object's content
- If not piggybacked: *retrieveRequired* = true and *dataContent* holds additional information about the data object for the file exchange protocol which handles the actual file transfer, e.g. a data identifier.

The second point needs some explanation: the actual data transfer not executed by SocioPath itself. Instead, the transfer needs to performed by a state-of-the-art file exchange protocol. That protocol should support downloads from multiple sources and can handle intermittent device unavailabilities. One example protocol that solves this problem is PPSPP . Here, the publishing device would initially seed the data object as a file. PPSPP would return a Merkle root hash as a unique file identifier which the publishing device can store in its Local Data Storage as a reference and send to other devices via the *dataContent* field in the NOTIFY$_{\text{Req}}$ message.

A device which receives the NOTIFY$_{\text{Req}}$ message, evaluates the value of *retrieveRequired*. If true, the Decision Engine is triggered accordingly with the **RetrieveRequired** event. The Decision Engine can now decide if, when and from what source(s) to retrieve the data object's content. Alternatively, if there are subscribed applications running on the device, the SODESSONretrieve method can be called by the application (see Section 4.3.6).

5.7.3 Data object retrievals

After a device a_1 has notified another device b_1 about a data object δ_t and δ_t was not piggybacked inside the NOTIFY$_{\text{Req}}$ message, b_1 can decide to retrieve it – either by an application request via SODESSON's retrieve method or due to an according Decision Engine's decision.

As soon as the Decision Engine decides to retrieve δ_t, it has to select one or more **source devices**. Source devices were inferred from previously received NOTIFY$_{\text{Req}}$ messages and stored in the *sources* field of the Local Data Storage entry.

The retrieval is requested by an RETRIEVE$_{\text{Req}}$ message which the requesting device sends to any known source device. This fields of this message are used to uniquely identify the requested data object.

The source device sends a RETRIEVE$_{\text{Rsp}}$ message with a boolean *requestAccepted* field. If true, the requesting device may initiate the data transfer with the responding device via the file exchange protocol. To this end, SocioPath passes the locator of the sources to that protocol plus any required information it has learned earlier from the dataContent field of a NOTIFY$_{\text{Req}}$ message.

5.7.4 Source management

The challenge that SocioPath has to solve is the selection of source devices – depending on the given privacy level, not all sources may be disclosed to subscribers. If privacy level IV (see Section

Table 5.17: *Additional fields inside a* RETRIEVE$_{Req}$ *message.*

Data Type	Field name	Description
`String`	topicTitle	Topic title of t ($title_t$)
`UID`	topicOwner	UID of the topic owner of t (UID_{TO_t})
`Integer`	objId	Object ID of δ_t (OID_{δ_t})
`Integer`	publishTime	Timestamp when data object was published. Used together with *objId* to uniquely identify the data object (see Section 4.3.1) and recognize obsolete notifications (see Section 4.3.7).

Table 5.18: *Additional fields inside a* RETRIEVE$_{Rsp}$ *message.*

Data Type	Field name	Description
`Boolean`	requestAccepted	if true: data transfer may be initiated

3.6) is targeted, i.e. topic owner A does not want subscriber B to know that C is also a member of the closed group and vice-versa. Thus, a topic owner device must pre-select the sources it announces to another device.

Source pre-selection by the topic owner

When a topic owner device needs to send a notification to a subscriber device, it pre-selects the sources it adds to each individual NOTIFY$_{\text{Req}}$ messages. It only announces devices that the subscriber device may learn about, depending on the privacy level:

- Privacy level III:
  - devices of the topic owner
  - devices of the publisher
  - devices of the all subscribers
- Privacy level IV:
  - devices of the topic owner
  - devices of the publisher
  - devices of the same subscriber

Announcing additional sources

The topic owner can only add devices it knows to be sources. At least the publishing device is one source.

As soon as a subscriber device has successfully retrieved the data content (or parts thereof, if the file exchange protocol supports it), it can be used as source for other devices. The topic owner must be informed about this.

Hence, after a successful retrieval the device simply has to send a notification about said data object to the topic owner devices and add itself to the *sources* field of the $\text{N}\scriptstyle{\text{OTIFY}}_{\text{Req}}$ message. The topic owner handles this notification like every other notification and sends new notifications to all other subscriber devices, each with pre-selected source as described above. Each device merges the new source with the existing sources with the entry in their Local Data Storage.

There are two optimizations possible here. First, a notification only needs to be sent to a subscriber device if the subscriber device may learn about the new source. Second, a notification only to needs to be sent to devices that are not sources yet, since new sources are irrelevant for devices that have already retrieved the data object.

An example is displayed in Figure 5.10 (a). Here, a_1 of topic owner A publishes a new data object and sends a $\text{N}\scriptstyle{\text{OTIFY}}_{\text{Req}}$ with a_1 (itself) as the only source to all subscriber devices (b_1, c_1, c_2). Then, c_2 sends a $\text{R}\scriptstyle{\text{ETRIEVE}}_{\text{Req}}$ to a_1 and the data transfer via PPSPP is initiated. c_2 now announces itself to the topic owner devices (here: a_1) as a new source. a_1 sends a new $\text{N}\scriptstyle{\text{OTIFY}}_{\text{Req}}$ to c_1, announcing (a_1, c_2) as the new sources.

In this case, A does not want B to learn that C is also a subscriber. When c_2 now announces itself as new source, a_1 announces this new source only to c_1, but not to b_1.

Unknown sources for the subscriber

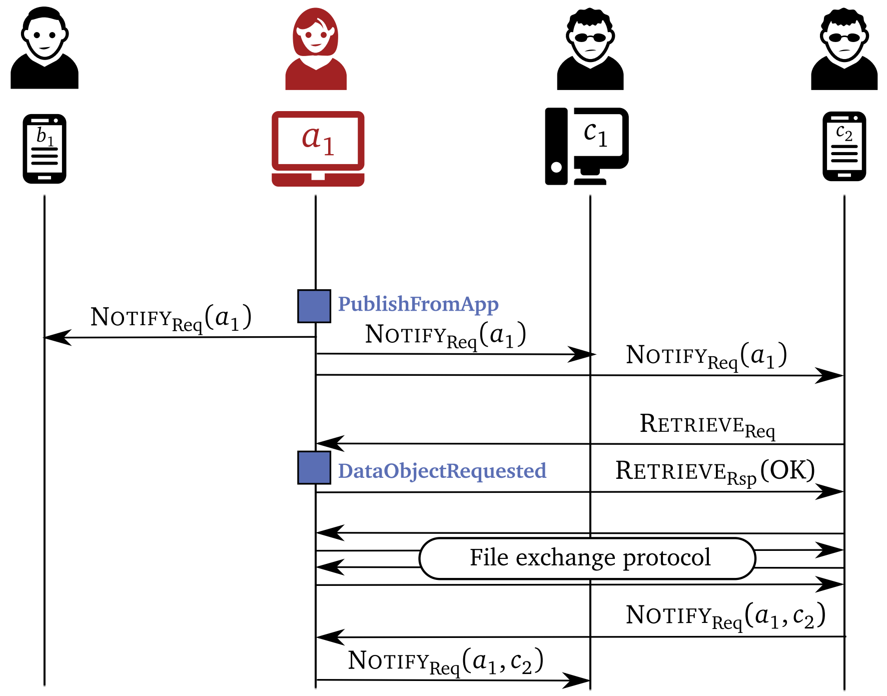
At least the publishing device is one source the topic owner can always announce. However, if publisher and subscriber are not contacts, the subscriber's device does not have an entry for the publishing device in its Device List. In this case, the subscriber requires at least one other source to retrieve the data object.

An example for this situation is also displayed in Figure 5.10 (b). Assume that B and C are not contacts. Note that A is unaware of this, since A does not know the Contact Lists of B and C. A accepts that B learns that C is also a member of the closed group and vice-versa. Therefore, topic owner device a_1 announces c_1 as a source to b_1. Since in SocioPath only devices of contacts can communicate with each other, the DID of c_1 is unknown to b_1, i.e. the announced source is useless for b_1. The same applies when a_1 announces c_2 to b_1. Only when a_1 has started the data content retrieval by itself and announces itself to b_1, b_1 has a valid source now.

5.8 Conclusion

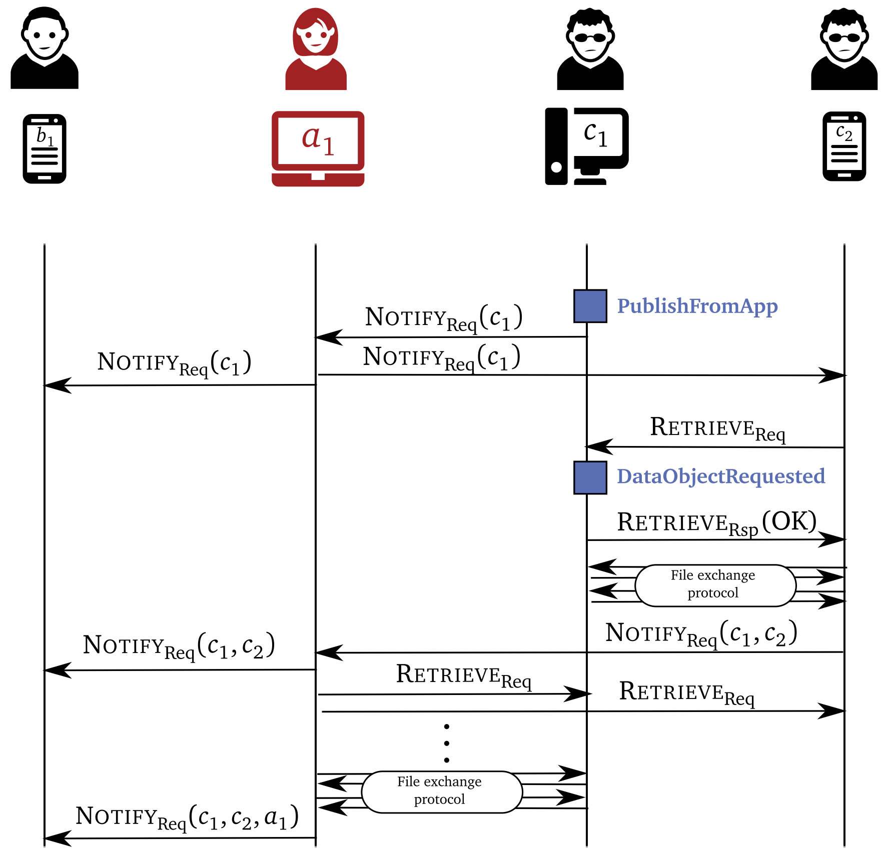
This chapter introduced SocioPath, a Data Dissemination Protocol (DDP) for the SODESSON middleware. Section 5.1 gave an overview on SocioPath's main features. These are realization of the central aspects in User-Centric Networking: self-sufficiency, partition tolerance and resource awareness. This chapter focused on the first of these three aspects and described how self-sufficient data exchange is generally enabled by SocioPath.

First, SocioPath's data structures were described in Section 5.2. As a basis for self-sufficiency, each device holds a list of all devices of all contacts and can thus communicate directly with them. Based on these data structures, the fundamentals of data object delivery in SocioPath was explained in Section 5.3. Afterwards, technical details about the protocol flow and message properties were discussed (Section 5.4) and further explained by revisiting the example of a



b_1
a_1
c_1
c_2
PublishFromApp
$\text{NOTIFY}_{\text{Req}}(a_1)$
$\text{NOTIFY}_{\text{Req}}(a_1)$
$\text{NOTIFY}_{\text{Req}}(a_1)$
$\text{RETRIEVE}_{\text{Req}}$
DataObjectRequested
$\text{RETRIEVE}_{\text{Rsp}}(\text{OK})$
File exchange protocol
$\text{NOTIFY}_{\text{Req}}(a_1, c_2)$
$\text{NOTIFY}_{\text{Req}}(a_1, c_2)$

(a) Topic owner publishes a data object where retrieve is required



b_1
a_1
c_1
c_2
PublishFromApp
$\text{NOTIFY}_{\text{Req}}(c_1)$
$\text{NOTIFY}_{\text{Req}}(c_1)$
$\text{NOTIFY}_{\text{Req}}(c_1)$
$\text{RETRIEVE}_{\text{Req}}$
DataObjectRequested
$\text{RETRIEVE}_{\text{Rsp}}(\text{OK})$
File exchange protocol
$\text{NOTIFY}_{\text{Req}}(c_1, c_2)$
$\text{NOTIFY}_{\text{Req}}(c_1, c_2)$
$\text{RETRIEVE}_{\text{Req}}$
$\text{RETRIEVE}_{\text{Req}}$
File exchange protocol
$\text{NOTIFY}_{\text{Req}}(c_1, c_2, a_1)$

(b) Non-topic owner publishes a data object where retrieve is required

Figure 5.10: *Topic owner announces new sources*

published blog post in SODESSON. With SocioPath, the device level which was not discussed and abstracted in Chapter 4), was complemented.

Section 5.6 described additional maintenance procedures for SocioPath, such as adding new devices and the message exchange for adding new contacts.

Section 5.7 explained the decoupling of notifications and data content retrievals, with the help of RETRIEVE messages in addition to NOTIFY messages. This complies with the decoupling of notifications and retrievals, which SODESSON already offers by its application interface. This decoupling is one of the building blocks for resource awareness in SocioPath and will be further discussed in Chapter 7 for Decision Engines.

Chapter 6

Partition Tolerance in SocioPath

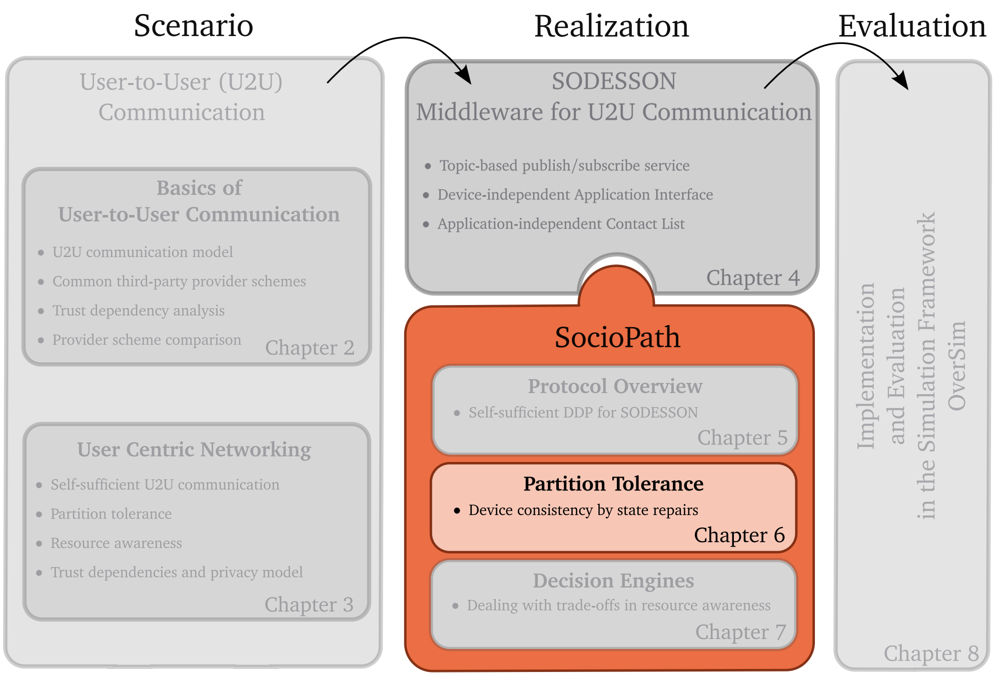

Scenario
Realization
Evaluation
User-to-User (U2U) Communication
Basics of User-to-User Communication
U2U communication model
Common third-party provider schemes
Trust dependency analysis
Provider scheme comparison
Chapter 2
User Centric Networking
Self-sufficient U2U communication
Partition tolerance
Resource awareness
Trust dependencies and privacy model
Chapter 3
SODESSON Middleware for U2U Communication
Topic-based publish/subscribe service
Device-independent Application Interface
Application-independent Contact List
Chapter 4
SocioPath
Protocol Overview
Self-sufficient DDP for SODESSON
Chapter 5
Partition Tolerance
Device consistency by state repairs
Chapter 6
Decision Engines
Dealing with trade-offs in resource awareness
Chapter 7
Implementation and Evaluation in the Simulation Framework OverSim
Chapter 8

Figure 6.1: *Placement of Chapter 6 in the big picture*

The basis for User-Centric Networking is the usage of the users' personal devices. These devices are heterogenous in terms of availability: a work PC might be switched off at night, network connectivity for a smartphone on a cell network could cut off if its owner drives through a tunnel, a network attached storage at home can be permanently connected to the Internet and rarely switched off. Hence, it is a basic task of a DDP to handle device unavailabilities. Self-sufficiency in SocioPath makes these conditions even more challenging, as there are no devices of third parties that can be leveraged.

As discussed in Chapter 5, a notification is sent to every device of a subscriber of the respective topic. If a device a has missed such a notification, it subsequently asks another device b about any missed notifications. b detects and resends notifications that b holds in its Local Data Storage and for which b knows that a is also supposed to receive, but a hasn't received these notifications so far. This procedure is called **state repair**.

The state repair procedure is based on the concept of **set reconciliation** [76]. Here, device a fills a data structure with the elements known to a. a sends this data structure to b. b infers the difference to a based on this data structure.

The challenge here lies in filling the data structure accordingly: depending on which user controls device a and b respectively, there are associated **relevant topics** and only the notifications for data objects with these relevant topics should be added to the respective data structure. Also, since the number of notifications in the Local Data Storage can grow over time, a space-efficient data structure is preferable.

Possible reasons for missed notifications are intermittent unavailability of the publishing device, a subscriber device or all topic owner devices at the time when an application on the publishing device published a new data object:

- *Publishing device unavailable:* Publishing device could not send the respective notification(s) to the topic owner device(s).
- *Subscriber device unavailable:* Subscriber device could not receive the respective notification when a topic owner device sent it to the subscriber devices
- *Topic owner devices unavailable:* Depending on the Decision Engine, the publishing device sends the respective notification(s) to either one or multiple topic owner devices. If these topic owner devices are unavailable, neither topic owner nor subscriber devices can receive the notification.

Until a state repair is performed, each device keeps working based on its current state, which is possibly out-of-date due to missed notifications. A specific case here is a topic owner device which performs access control based on its current Own Topics List. Updates are assumed to propagate eventually to formerly unavailable devices via state repairs, similar to the concept of *eventual consistency* [115].

Set reconciliation is described in Section 6.2. The actual state repair procedure, including the required message exchanges, is described in Section 6.3.

6.1 Consistency Demands for SocioPath

The section discusses the consistency demands for SocioPath. It deals with the question what kind of synchronization between two devices must be achieved by a state repair.

First, a distinction between notifications and data content needs to be made. Given are one device a of user A and one device b of user B:

- **Notifications:** a and b are consistent with regard to notifications if: for each data object that is **relevant** to A and B, both a and b hold notifications in their Local Data Storage.

Notifications for data objects that are not relevant to both A and B do not count towards consistency between a and b.

- **Maintenance data objects:** a and b are consistent in regard to maintenance data objects if: for each maintenance data object *relevant* to A and B, both a and b hold the same maintenance data object in their Local Data Storage. Maintenance data objects that are not relevant to both A and B do not count towards consistency between a and b.

- **Application data content:** Application data content does not count towards consistency. Each device decides for itself whether to retrieve application data content, which might never happen. Therefore, if device a has retrieved a data object δ_t and b has not, they are inconsistent from the application's point of view, but they might still be consistent in SocioPath.

As discussed in Section 4.3.1, in SODESSON each data object δ_t is uniquely identified by a 3-tuple (topic t, object ID OID_{δ_t}, creation/update timestamp $time_{\delta_t}$). This 3-tuple defines whether two devices hold the same notifications.

6.1.1 Relevant Topics

The notifications and maintenance data objects which device a of user A and device b of user B must hold to be consistent, differ with each combination of A and B. All of these notifications and maintenance data objects are associated to topics which are relevant to A and B.

A topic t which is **relevant** to A and B fulfills one of the following conditions:

- A is topic owner of t and B is subscriber of t
- A is topic owner of t and B is allowed publisher for t
- $A = B$

Analogously, a *relevant notification* is a notification in regard to a relevant topic. A *relevant maintenance data object* is a maintenance data object in regard to a relevant topic.

The first condition covers the case where a is a device of topic owner A and b is a device of subscriber B. Topic owner device a holds notifications about data objects with topic t and subscriber device b should receive notifications about topic t. a and b aim to be consistent here. Therefore these notifications have to be synchronized from a to b. For maintenance data objects, the condition holds in an analogous way: Here, maintenance data objects where A is the topic owner and B is subscriber (e.g. {"Devices", A}) have to be synchronized from a to b.

The second condition covers the opposite direction: If a is a device of topic owner A and b is a device of allowed publisher B, b might have published new data objects that topic owner device a should know about. Therefore these notifications have to be synchronized from b to a. For maintenance data objects, the condition holds in an analogous way: Here, maintenance data objects where A is topic owner and B is an allowed publisher (e.g. {"SubscribeMe", A}) have to be synchronized from b to a.

A special case here is consistency between two devices of the same user, i.e. $A = B$. Here, all topics are relevant that A is subscriber, topic owner or allowed publisher of. The aim here is that all devices hold exactly the same notifications and maintenance data objects to be fully consistent.

Example

Two examples shall illustrate the previous discussion. Assume topic owner A has two topics t_1 and t_2. For t_1, the users B and C are both subscribers and allowed publishes. For t_2, only B is subscriber and an allowed publisher, C is neither. Each user has one device, i.e. there are a, b and c respectively.

In Table 6.1, all three devices start with empty Local Data Storages and are thus consistent (line 1).

In line 2, b publishes a new data object t_1 with objId 42 at time 100, i.e. the 3-tuple (t_1, 42, 100) identifies the data object. At this time, b is not consistent with topic owner device a anymore, since b has not yet sent a notification to a. However, a and c are still consistent since their Local Data Storages are still empty.

In line 3, topic owner device a was notified about the new data object, thus b and a are consistent, but a and c are not consistent anymore.

In line 4, topic owner device a has notified subscriber device c about the new data object, thus a and c are now also consistent.

In line 5, c overwrites the data object by publishing a new data object at time 150 with the same topic t_1 and objId 42, i.e. the 3-tuple (t_1, 42, 100) identifies the new data object. Now a and c are not consistent anymore, since a still holds the notification about the data object that was overwritten. However, b and a are still consistent from their point of view.

In line 6, topic owner device a was notified about the new data object, thus c and a are consistent, but a and b are not consistent anymore.

Finally, in line 7 topic owner device a has notified subscriber device b about the new data object.

Table 6.1: *Consistency between devices: first example*

Device b			**Topic owner device a**			Device c			(b,a) cons.	(a,c) cons.
topic	objId	time	topic	objId	time	topic	objId	time		
									✓	✓
t_1	42	100							✗	✓
t_1	42	100	t_1	42	100				✓	✗
t_1	42	100	t_1	42	100	t_1	42	t_1	✓	✓
t_1	42	100	t_1	42	100	t_1	42	**150**	✓	✗
t_1	42	100	t_1	42	**150**	t_1	42	**150**	✗	✓
t_1	42	**150**	t_1	42	**150**	t_1	42	**150**	✓	✓

Table 6.2 shows a similar situation, however this time a publishes a data object with topic t_2 where C is no subscriber. Thus, the data object is not relevant to A and C. Thus, a and c are still consistent.

Table 6.2: *Consistency between devices: second example*

Device b			**Topic owner device a**			Device c			(b,a) cons.	(a,c) cons.
topic	objId	time	topic	objId	time	topic	objId	time		
									✓	✓
t_2	23	180							✗	✓
t_2	23	180	t_2	23	180				✓	✓

6.1.2 Positioning SocioPath in the CAP Model

In order to motivate state repairs as suitable for missed notifications, SocioPath shall now be positioned into the CAP model. For the CAP model, Brewer [14] named three system requirements that are inherent to each service provided by a distributed system: consistency, availability and partition tolerance (or short: CAP).

- **Consistency:** the service ensures that all instances share the same view at all times. Gilbert and Lynch compared a consistent service to one atomic data object. For this data object, "*there must exist a single order of operations such that each operation looks as if it were [...] executing on a single node, one operation at a time.*" [40]

- **Availability:** for each service request, a service response is given

- **Partition tolerance:** the service allows for the situation where two instances are unable to communicate with each other

Brewer stated that it is impossible for a distributed system to provide all three, i.e. there are only CA, AP or CP systems. This conjecture was later formally proven by Gilbert and Lynch [40] and is known as the CAP theorem.

Here, the service in the CAP model is the publish/subscribe service for data objects that SODESSON offers to applications for U2U communication. Since SocioPath is a DDP for SODESSON and responsible for all inter-device communication, SocioPath's system properties also hold for the publish/subscribe service.

In Section 2.1, U2U communication was defined to be *delay-tolerant*, i.e. to have no real-time constraints. In terms of the CAP theorem, this means that the aspect of consistency (C) can be neglected. To see this, assume that device a of user A publishes a new data object δ_t and device b of user B is a subscriber device. If consistency (C) was essential here, b would have to retrieve δ_t instantly, so both devices share a consistent view. According the to the CAP theorem, this would mean that SocioPath is either a CP system (where availability is disregarded) or a CA system (where partition tolerance is neglected). For a pub/sub service for U2U communication, enabled by user-controlled devices, both variants are not acceptable:

- **CP (disregarding availability)**: Due to the partition tolerance, the case where devices a and b are unable to communicate may occur. However, in that case a is not allowed to publish δ_t, i.e. the pub/sub service itself as provided by SODESSON is not available. More generally, the pub/sub service is only available if all devices of all subscribers are available at the same time. This could be acceptable if B was the only subscriber and b the only subscriber device, but in case of multiple subscriber devices, one unavailable subscriber device disrupts the pub/sub service for all other subscriber devices.
- **CA (disregarding partition tolerance)**: The pub/sub service should be available at all times, so both devices a and b need to be highly available and able to communicate at any time in order to maintain consistency between them. This cannot be applied to users' personal devices where intermittent unavailability is expected.

Instead, SocioPath gets classified as an AP system, where consistency (C) is neglected. This means that inconsistencies between device a and b are expected: If devices a and b are in different partitions and therefore a cannot notify b about δ_t yet, this is acceptable due to delay-tolerance of U2U communication. This temporary inconsistency is accepted in favor of an application running on a being able to publish at any time and to deliver δ_t to other subscriber devices if possible.

State repairs shall eventually remove inconsistencies between any two devices, however no time guarantees are made.

6.2 Set Reconciliation

State repairs are based on the idea of set reconciliation, which is defined and explained in this section.

6.2.1 General Problem Definition

Assume two generic instances A and B. Each instance is holding a set of items S_A and S_B respectively (see Figure 6.2a). The problem to be solved now is B recognizing the difference of the sets $S_B - S_A$ and providing A with that difference, i.e. A should hold $S'_A = S_A \cup (S_B - S_A) = S_A \cup S_B$. In order that B can infer the difference $S_B - S_A$, A has to send a data structure which holds S_A to B first (see Figure 6.2b). B then sends the difference based on this information (see Figure 6.2c). If this procedure is repeated vice-versa, i.e. by A sending the difference $S_A - S_B$ to B (see Figures 6.2d to 6.2f)), both A and B hold the same set of items $S'_A = S'_B = S_A \cup S_B$ (see Figure 6.2g).

6.2.2 Data Structures for Set Reconciliation

This section discusses two possibilities for the data structures holding notifications for set reconciliation in SocioPath. Given this data structure has to hold an arbitrary number of notifications, this data structure is preferably space-efficient. Two different data structures are discussed as follows.

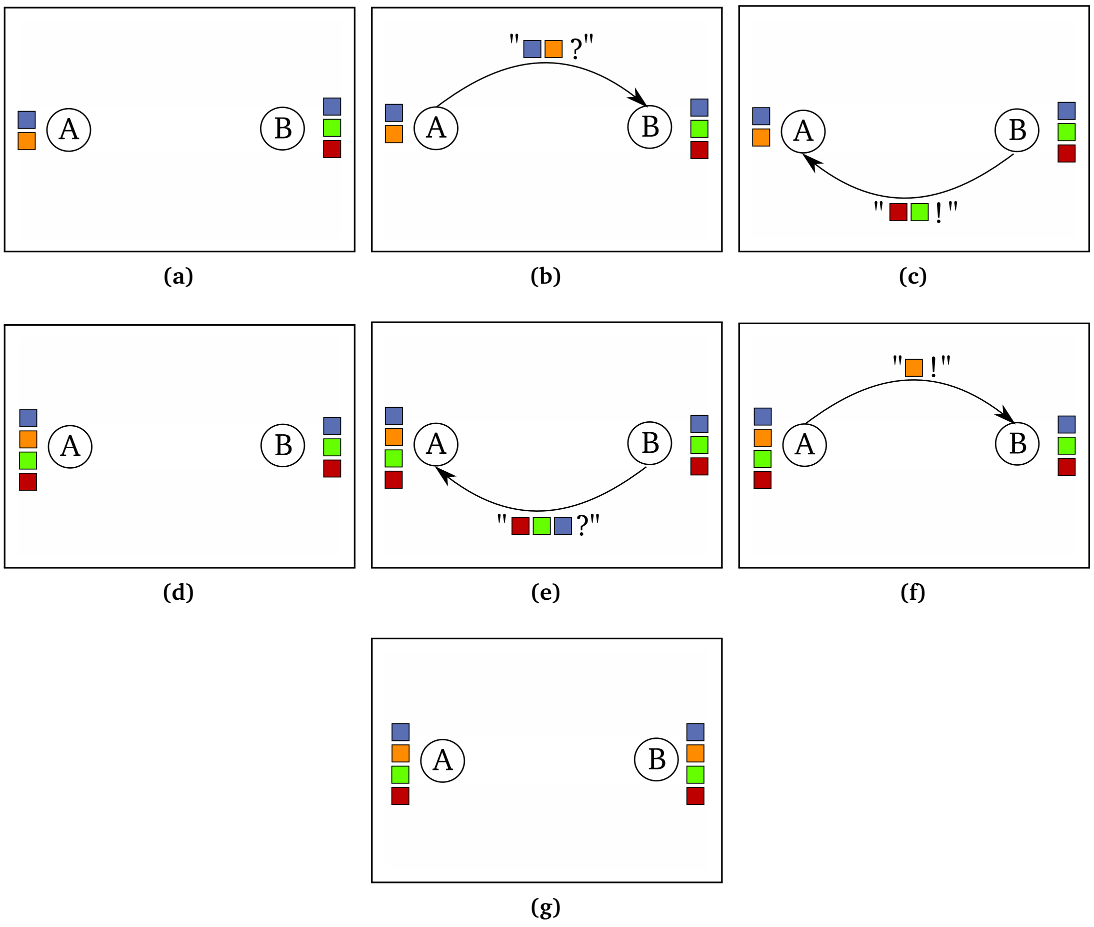


Figure 6.2: *Set reconciliation between two generic instances*

6.2.2.1 Identifier List

The first regarded possible data structure is a simple identifier list. As discussed in Section 4.3.1, each data object δ_t is uniquely identified by a 3-tuple:

- Topic t
- Object ID OID_{δ_t}
- Creation / update timestamp $time_{\delta_t}$

The most simple data structure for sending known notifications is a list of such 3-tuples. However, this is not space-efficient: As defined in Section 4.3.1, a topic consists of a title $title_t$ (a string of arbitrary length) plus the User ID TO_t of the topic owner. Assuming the following sizes, this results in 32 bytes plus the bytesize for $title_t$ per notification:

- TO_t: 160 bits (e.g. SHA-1 hash)

- OID_{δ_t}: 64 bits
- $time_{\delta_t}$: 32 bits (e.g. Unixtime)

Since the title string $title_t$ can be freely defined by the application, it is difficult to make an estimation here. Real world applications would probably follow hierarchical naming conventions, similar to Java package names (i.e. "*com.mycompany.department.application*") where 40-50 characters are easily reached. Assuming an UTF-8 encoding with 1-2 bytes per character, this would result in up to 100 bytes per $title_t$, plus the 32 bytes as suggested above, resulting in a total 132 bytes per notification.

6.2.2.2 Bloom Filters

Broder and Mitzenmacher [15] have formulated the *Bloom filter principle*: "*Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.*"

Bloom filters [13] are a probabilistic, space-efficient data structure which represents a set $S := \{x_1, x_2, \dots, x_n\}$ of n elements. Each element $x \in S$ is hashed with k independent hash functions $h_i(x) : x \to \{1 \dots m\}$ to an array of bits with length m. The bits $h_i(x)$ are set from 0 to 1 for $1 \leq i \leq k$.

This allows queries if a specific element y is member of S, given that the k hash functions are known to the enquirer: If $h_i(y) = 1$ for all $1 \leq i \leq k$, then it is assumed that $y \in S$. However, false positives may occur with a certain false positive rate p which depends on k, m and n. An approximation for the false positive rate p is:

$$p = (1 - e^{-\frac{k*n}{m}})^k \tag{6.1}$$

Note: False negatives cannot occur, i.e. the result $y \notin S$ is always reliable.

Impact of false positives

In [15], *approximate set reconciliation* is named as one application for Bloom filters in P2P networks. It is only approximate due to the effect of false positives. In the context of set reconciliation, a false positive means that with a certain probability p, A falsely detects an element $x \in S_A, x \notin S_B$ to be $x \in S_A \cap S_B$. Therefore, A does not add x to the difference set which A returns to B and B never learns about the existence of x.

The usefulness of such an approximate set reconciliation depends on the chosen false positive rate. Given that the false positive rate is small enough, it can be neglected. For comparison, hard disk vendor specificiations predict an uncorrectable bit read error probability of 10^{-16} to 10^{-13} [48].

Size estimation

The false positive rate depends on three parameters: the number of hashes k, the number of elements n and the field size of the Bloom filter, i.e. the number of bits, m.

The k hashes need to be agreed upon between the two instances. In a real-world implementation, this can be realized by A sending k seed values in addition to the bloom filter data structure, which results in an overhead $o(k)$ whose size depends on k. *bloom* [12], a Bloom filter C++ library, represents each seed value as a 32 bit integer, i.e. $o(k) = k * 4$ bytes can be used as a rule of thumb here.

Therefore, if a specific false positive rate $\bar{p}$ is targeted and the number of elements n is known a priori and a number k is chosen, the required bloom filter size m can be inferred by transforming Equation 6.1:

$$m = -\frac{k * n}{ln(1 - \sqrt[k]{\bar{p}})}[\text{bits}] \tag{6.2}$$

In summary, instance A has to send $z = m + o(k)$ bits to instance B. Since both k and m influence the false positive rate for a given n, z can be minimized as an optimization problem, i.e. by finding optimal values for k and m.

6.2.2.3 Size comparison

The two regarded data structures are now compared in their sizes for a different number of elements, based on the previous size estimations. For the Bloom filters, a fix value for $k = 32$ is assumed, therefore $o(k) = 4 * 32 = 128$ bytes has to be added as a fix overhead.

The results are displayed in Table 6.3. As can be seen there, the Bloom filter outperforms the identifier list by about factor 10 in term of space-efficiency, with a negligible false positive rate of 10^{-16}. For a small number of elements, the gain is smaller, due to the fix $o(k) = 128$ bytes. This can be further reduced by solving the optimization problem and finding optimal values for k and m.

Table 6.3: *Size comparision: Identifier list vs Bloom filter*

Data structure	Number of elements	False positive rate	Estimated space
Identifier List	10	0.0	1320 Bytes
	100	0.0	13.200 Bytes
	1000	0.0	132.000 Bytes
	10.000	0.0	1.320.000 Bytes
Bloom Filter ($k = 32$)	10	10^{-16}	234 Bytes
	100	10^{-16}	1181 Bytes
	1000	10^{-16}	10.651 Bytes
	10.000	10^{-16}	105.356 Bytes

6.3 State Repairs

State repairs can be performed between two devices a and b as soon as they are in the same network partition and are thus available for each other.

A state repair of device a by device b is performed by three steps:

(1) Device a sends a $\text{STATE}_{\text{Req}}$ message to device b. This message contains a Bloom filter which holds notifications known to a and relevant to the devices' users.

(2) Device b matches the relevant notifications b knows from its own Local Data Storage against the Bloom filter. It detects the set of missed relevant notifications $\mathcal{M}$ and returns a $\text{STATE}_{\text{Rsp}}$ message to device a. This message contains the size of $\mathcal{M}$, i.e. the number of missed relevant notifications that b has detected.

(3) Device b sends the $|\mathcal{M}|$ missed notifications as $\text{NOTIFY}_{\text{Req}}$ messages to a.

Note that this an unidirectional state repair, i.e. a initiates the state repair by sending the $\text{STATE}_{\text{Req}}$ message to b in order to have a's state repaired by b (as much as possible for b). In order for b to be repaired by a, b must analogously send a $\text{STATE}_{\text{Req}}$ message to a.

Application and maintenance data content from the Local Data Storage gets only transferred during this procedure if the DE on b decides to piggyback them in the notifications. Since relevant maintenance data content also count towards consistency between two devices, they either have to be piggybacked in the notifications from b or completely retrieved by a until full consistency between a and b is achieved.

A special case of former device unavailability is adding a new device to a user's device pool. In order to transfer notifications from an existing device to a new device, the state repair mechanism can be used: The situation where a new device appears is essentially identical to the device being unavailable forever and now the complete Local Data Storage needs to be repaired.

In order to reduce redundancy in the exchanged information, each device keeps track of the time when its last successful state repair was performed with another specific device. Inside the next $\text{STATE}_{\text{Req}}$ message, a sends an according timestamp to b and both devices only regard notifications they have received since that time. For a proper functioning, all devices are assumed to be synchronized to a global wallclock, for example via the NTP protocol [75].

6.3.1 Reactive vs. Periodic State Repairs

Sending a $\text{STATE}_{\text{Req}}$ message from device a is triggered by the Decision Engine event **SendState**. The event can occur for different reasons, two of them are regarded here and will be important for the evaluation (Chapter 8).

In the most simple case, the Decision Engine on each device sets a periodic timer, so each device sends its own $\text{STATE}_{\text{Req}}$ messages regularly and **periodically**. The period is a system-wide parameter called the State Sending Interval (SSI).

As alternative, state repairs can be triggered **reactively**: if a device gets switched on or regains formerly missing availability, it initiates state repairs. This would require a callback from the device's operating system to SocioPath in order to signal such a low-level event. In the reactive case, the device also demands a returned $\text{STATE}_{\text{Req}}$ from the contacted devices. The latter is required for the case that the device has published new data objects during its unavailability and now needs to send the relevant notifications. This demand is not required for periodic state repair, since each device sends its own $\text{STATE}_{\text{Req}}$ messages regularly. However, reactive state repairs strongly depend on each device's individual availability and no assumptions about other devices are made here.

6.3.2 Message Exchange

On the Decision Engine event **SendState**, *a*'s Decision Engine has to decide which users (i.e. a subset of the Contact List) are to be targeted. For each state *target user*, the content of the $\text{STATE}_{\text{Req}}$ message is different, since the relevant topics differ from target user to target user. Then, for each target user, the Decision Engine has to decide which state *target devices* the $\text{STATE}_{\text{Req}}$ message has to be sent to and in what order: it might be a good idea to target a device with a track record of high availability and low communication costs first, since this device can probably already cover most of the missed notifications and thus take the load off other target devices of the same target user. According to this decision, a sends $\text{STATE}_{\text{Req}}$ message(s) with the fields displayed in Table 6.4 to target devices.

For the remainder of this section, it is always assumed that device a owned by user A sends a $\text{STATE}_{\text{Req}}$ message to device b owned by user B. First of all, a iterates over its Local Data Storage and collects all notifications tied to relevant topics $\mathbb{R}_{A,B}$. For each collected notification, the 3-tuple <*topic*, *objectId*, *publishTime*> is hashed into a Bloom filter data structure, which is later the field *bloomFilter* in the $\text{STATE}_{\text{Req}}$ message. The pseudocode for building a Bloom filter in regard to users A and B is displayed in Algorithm 1.

Table 6.4: *Additional fields inside a* STATE_{Req} *message.*

Data Type	Name	Explanation
bloom	bloomFilter	Efficient data structure for holding notifications from the requesting device's Data Storage
unsigned int	lastOkStateTime	Timestamp when last state repair by target device was successful. 0 if target device has never performed a successful state repair.
bool	demandTargetState	Ask target device to send $\text{STATE}_{\text{Req}}$ message as well – used for reactive state repairs

Table 6.5: *Additional fields inside a* STATE_{Rsp} *message.*

Data Type	Name	Explanation
unsigned int	numMissed	number of missed notifications ($\lvert\mathcal{M}\rvert$)

A device that has received a $\text{STATE}_{\text{Req}}$ message shall always directly respond. Thus, as soon as b receives the $\text{STATE}_{\text{Req}}$ message, it collects all notifications $\mathcal{N}$ from its Local Data Storage that are relevant for a and have arrived at b since *lastOkStateTime*. For each collected notification in $\mathcal{N}$, b checks if the tuple <*topic*, *objectId*, *publishTime*> was hashed by a into the *bloomFilter* data structure. If b cannot match the tuple to the Bloom filter, b assumes that the respective notification was missed by a. Hence, b collects a subset $\mathcal{M} \subseteq \mathcal{N}$ of assumably missed notifications which are to be resent.

Algorithm 1 Building Bloom filters $\text{STATE}_{\text{Req}}$ message to target user B

```
 1: procedure BUILDSTATEFORUSER($X \in \mathbb{U}$)
 2:     BloomFilter bloomFilter $\leftarrow \perp$
 3:     Topic[ ] relevantTopics $\leftarrow \perp$
 4:     $A \leftarrow$ getOwnUserId()
 5:     for each $t \in$ SubscriptionList do
 6:         if $t$.getTopicOwner() $= B$ then            ▷ relevant topics, 1st condition
 7:             relevantTopics.add($t$)
 8:     for each $t \in$ OwnTopicsList do
 9:         if $B \in \mathbb{P}_t$ then                  ▷ relevant topics, 2nd condition
10:             relevantTopics.add($t$)
11:     if $A = B$ then                                   ▷ special case: A = B
12:         for each $t \in$ LocalDataStorage do
13:             relevantTopics.add($t$)
14:     for each $t \in$ relevantTopics do
15:         for each $\gamma \in d$.LocalDataStorage.get($t$) do
16:             bloomFilter.insert($objId_\gamma, time_\gamma$)
17:     return bloomFilter
```

Subsequently, b replies with a $\text{STATE}_{\text{Rsp}}$ message which contains $|\mathcal{M}|$ in the *numMisses* field, so a knows how many $\text{NOTIFY}_{\text{Req}}$ resends it has to expect for the state repair to be successful. Finally, b resends the $\text{NOTIFY}_{\text{Req}}$ messages to a. The resent notifications' contents are identical to an original notification, as described in Section 5.7.2 and are processed by a accordingly, with one difference: the field *stateNonce* contains the *msgNonce* from the original $\text{STATE}_{\text{Req}}$ message. This is information needed for a to identify the incoming $\text{NOTIFY}_{\text{Req}}$ messages as resendings triggered by the $\text{STATE}_{\text{Req}}$ and count them towards the $|\mathcal{M}|$ expected $\text{NOTIFY}_{\text{Req}}$ messages as announced by the $\text{STATE}_{\text{Rsp}}$ message.

Since a Bloom filter is based on hashes, it only allows to check whether it contains a specific notification, but the previously inserted notification cannot be extracted afterwards. This is the reason why each state repair is unidirectional: b cannot infer from a's $\text{STATE}_{\text{Req}}$ message whether a knows any notifications that are relevant for b. To learn this, b has to send its own $\text{STATE}_{\text{Req}}$ message to a. However, a can ask b to do so by setting the in *demandTargetState* flag to true its own $\text{STATE}_{\text{Req}}$ message.

6.3.3 Successful vs. Unsuccessful State Repairs

A state repair of device a by device b was successful if and only if

(1) Device a received as many resent notifications from b as b has announced in its $\text{STATE}_{\text{Rsp}}$ message

(2) These resent notifications have the same *stateNonce* as the *msgNonce* of a's $\text{STATE}_{\text{Req}}$ message.

Device a can check both conditions and therefore infer if its latest state repair by b was successful or not.

Reasons for unsuccessful state repairs are:

- b did not receive a's $\text{STATE}_{\text{Req}}$ message
- a did not receive b's $\text{STATE}_{\text{Rsp}}$ message
- a did not receive all notifications that were announced in the $\text{STATE}_{\text{Rsp}}$ message

The timeout by which time a needs to have received all the notifications announced in the $\text{STATE}_{\text{Rsp}}$ message is the next state repair: If by the next time when a initiates a state repair by device b, the previous state repair was not closed as successful, then it is assumed as unsuccessful.

The knowledge whether a state repair was successful or unsuccessful, is used for improving space effienciency, as discussed in the next section.

6.3.4 Improving space efficiency

Local Data Storages can grow large over time. This results in larger Bloom filters and therefore larger $\text{STATE}_{\text{Req}}$ messages that the devices which are possibly resource-constrained have to send and receive. In order to improve space efficiency, state repair has an additional mechanism.

The mechanism is based on the idea that successful state repairs in the past have established consistency between the two devices. Therefore, all notifications that were received by either device before the last successful state repair can be ignored. In other words, only notifications that were received by either device since the last successful state repair need to be taken into account. This narrows the time window in question from "all relevant notifications ever" to "relevant notifications since the last successful state repair until now".

For each device x in its Devices List, a saves the timestamp *lastOkStateTime* when the last successful state repair by x was performed. This value for b is also entered into $\text{STATE}_{\text{Req}}$ message sent from a to b. The value equals 0 if the target device has never performed a successful state repair.

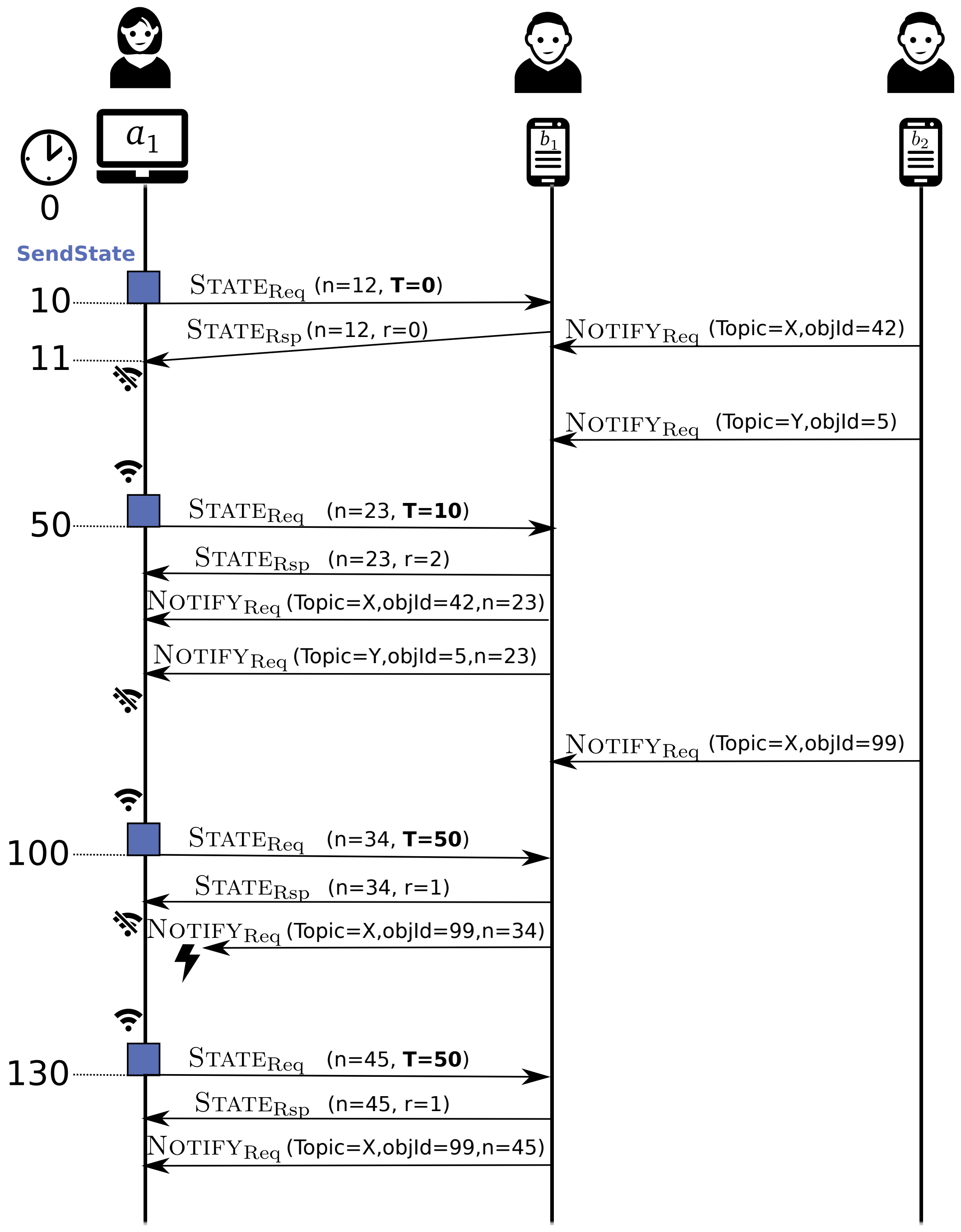
Additionally, a only hashes the relevant notifications it has received since *lastOkStateTime* into the Bloom Filter, as described above, i.e. where *gotNotifyTime* from the Local Data Storage entry is newer than *lastOkStateTime*. b on the other hand, only compares the relevant notifications it has received since the *lastOkStateTime* passed by a in the $\text{STATE}_{\text{Req}}$ message.

6.3.5 Example

Figure 6.3 displays an example sequence where device a_1 owned by user A sends $\text{STATE}_{\text{Req}}$ messages to device b_1 owned by user B. The figure covers three different outcomes of the state repair which are described as follows.

No state repair required

Wallclock time 10: a_1 sends a $\text{STATE}_{\text{Req}}$ message to b_1. This $\text{STATE}_{\text{Req}}$ message has its *lastOk-StateTime* field set to $T = 0$, since b_1 has never successfully repaired a's state before. Hence, b_1 collects $\mathcal{N}$, i.e. all notifications with a relevant topic and *GotNotify* timestamp greater or equal than $T = 0$ from its Local Data Storage. b_1 checks if the received Bloom filter contains



a_1
b_1
b_2
0
SendState
10
11
50
100
130
STATE$_{Req}$ (n=12, **T=0**)
STATE$_{Rsp}$ (n=12, r=0)
NOTIFY$_{Req}$ (Topic=X,objId=42)
NOTIFY$_{Req}$ (Topic=Y,objId=5)
STATE$_{Req}$ (n=23, **T=10**)
STATE$_{Rsp}$ (n=23, r=2)
NOTIFY$_{Req}$ (Topic=X,objId=42,n=23)
NOTIFY$_{Req}$ (Topic=Y,objId=5,n=23)
NOTIFY$_{Req}$ (Topic=X,objId=99)
STATE$_{Req}$ (n=34, **T=50**)
STATE$_{Rsp}$ (n=34, r=1)
NOTIFY$_{Req}$ (Topic=X,objId=99,n=34)
STATE$_{Req}$ (n=45, **T=50**)
STATE$_{Rsp}$ (n=45, r=1)
NOTIFY$_{Req}$ (Topic=X,objId=99,n=45)

🕒	Wallclock time
⊘	Device becomes available
◠	Device becomes unavailable

Figure 6.3: *Example message exchanges for state repair*

all notifications in $\mathcal{N}$. This is the case here, so b_1 sets the *numMisses* field $r = 0$ in the $\text{STATE}_{\text{Rsp}}$ message.

Wallclock time 11: On receiving the $\text{STATE}_{\text{Rsp}}$ response, a_1 learns that no state repair is required by b_1 and sets the *lastOkStateTime* for b_1 in the Devices List to $T = 10$. Note that the new *lastOkStateTime* value $T = 10$ equals the time where a sent the $\text{STATE}_{\text{Req}}$ message, not $T = 11$, where it received the $\text{STATE}_{\text{Rsp}}$ message.

Figure 6.3 shows the reason for this: After sending the $\text{STATE}_{\text{Rsp}}$ response, b_1 receives a new $\text{NOTIFY}_{\text{Req}}$ from b_2 which is also relevant for a. This happens before wallclock time 11. However, a becomes unavailable directly after the state message exchange with b_1, thus b_1 cannot successfully send the $\text{NOTIFY}_{\text{Req}}$ to a_1. If a_1 would set $T = 11$ in its next $\text{STATE}_{\text{Req}}$ message, b_1 would not take this notification into account and a_1 would never learn about it.

Successful state repair

Wallclock time 50: After a becomes available again, the DE decides to initiate state repair with b_1. a_1 sends a new $\text{STATE}_{\text{Req}}$ message to b_1, this time with $T = 10$ (as discussed above). b_1 sees that there it received two new notifications since $T = 10$ that the Bloom filter does not contain. Hence, b_1 sets the *numMisses* field $|\mathcal{M}| = 2$ in the $\text{STATE}_{\text{Rsp}}$ message and resends both notifications to a_1. Both $\text{NOTIFY}_{\text{Req}}$ messages have their *stateNonce* field n set to the *nonce* value of the original $\text{STATE}_{\text{Req}}$ message. This way, a_1 can differentiate the resends from new notifications and hence expects two different notifications with $n = 23$. After receiving the two notifications, the state repair is successful. a_1 sets the *lastOkStateTime* for b_1 in the Devices List to $T = 50$. a_1 again becomes unavailable directly afterwards.

Unsuccessful or partial state repair

Wallclock time 100: Once again, a_1 becomes available. At first, a_1 and b_2 act in an analog way as before at wallclock time 50, except for a new $\text{STATE}_{\text{Req}}$ *nonce* $n = 34$ and $T = 50$. After wallclock time 50, b_1 has received one notification relevant for a_1. Hence, b_1 sets the *numMisses* field $|\mathcal{M}| = 1$ in the $\text{STATE}_{\text{Rsp}}$ message and resends the notification to a_1. However, before receiving the resent notification, a becomes unavailable again. Since a did not receive $|\mathcal{M}| = 1$ notifications with $n = 34$ in the *stateNonce* field, it does not regard the state repair as successful and thus does not increase T.

Wallclock time 130: When a comes back online and the DE initiates a new state repair, it is still $T = 50$. Hence, b resends the notification that was not successfully received by a during the last state repair.

6.4 Conclusion

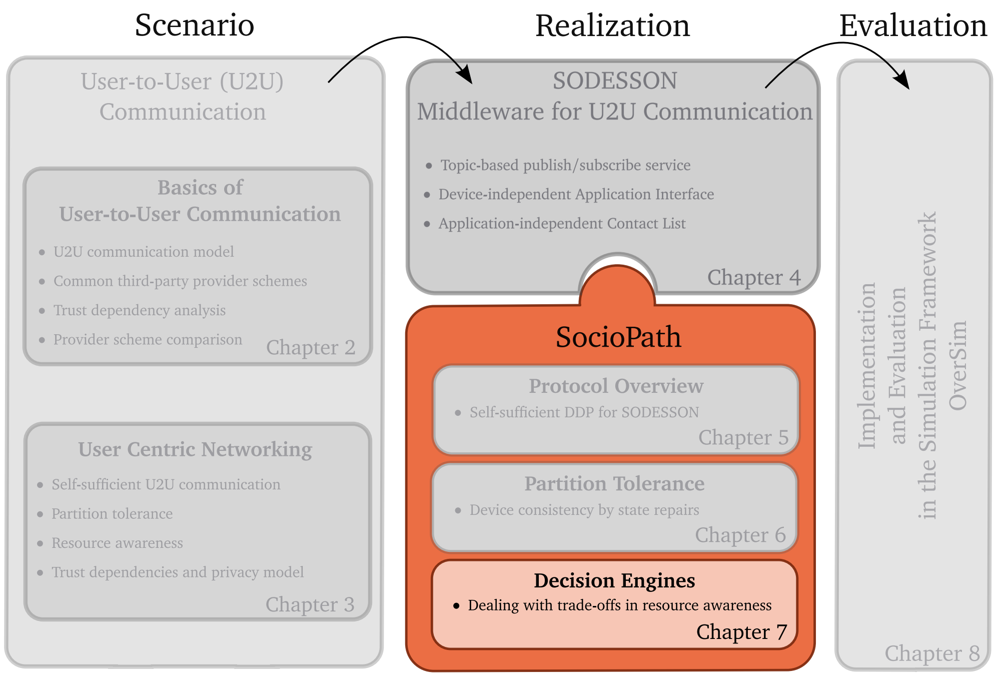
This chapter presented partition tolerance for a discussion on partition tolerance in User-Centric Networking in SocioPath, i.e. how SocioPath copes with intermittent unavailabilites of personal devices.

SocioPath achieves partition tolerance by the process of set reconciliation (Section 6.2) of relevant notifications and maintenance data objects. In order to make set reconciliation more space-efficient, Bloom filters are regarded as one possible data structure. Bloom filter outperforms a simple identifier list by about factor 10 in term of space-efficiency, with a negligible false positive rate of 10^{-16}.

Section 6.3 presented the state repair mechanism, which is the set reconciliation of notifications in two devices' respective Local Data Storage. This way, two devices resynchronize on their state after a previous unavailability. Depending on the users of the regarded devices and their roles (publisher, subscriber, topic owner), different notifications need to be exchanged. These relevant notifications are identified by their topic, i.e. there are different relevant topics depending on the constellation of the affected users' roles. The respective notifications are hashed into a Bloom Filter, which is sent via a $\text{STATE}_{\text{Req}}$ message. For additional space efficiency, each device remembers the timestamp of each last successful state repair for all other known devices. Any notifications older than that timestamp can be disregarded for the next state repair and do not need to be hashed into the Bloom Filter of the next $\text{STATE}_{\text{Req}}$ message.

Chapter 7

SocioPath: Decision Engines



Scenario
Realization
Evaluation
User-to-User (U2U) Communication
Basics of User-to-User Communication
U2U communication model
Common third-party provider schemes
Trust dependency analysis
Provider scheme comparison
Chapter 2
User Centric Networking
Self-sufficient U2U communication
Partition tolerance
Resource awareness
Trust dependencies and privacy model
Chapter 3
SODESSON Middleware for U2U Communication
Topic-based publish/subscribe service
Device-independent Application Interface
Application-independent Contact List
Chapter 4
SocioPath
Protocol Overview
Self-sufficient DDP for SODESSON
Chapter 5
Partition Tolerance
Device consistency by state repairs
Chapter 6
Decision Engines
Dealing with trade-offs in resource awareness
Chapter 7
Implementation and Evaluation in the Simulation Framework OverSim
Chapter 8

Figure 7.1: *Placement of Chapter 7 in the big picture*

Chapter 5 described how data objects are generally delivered in SocioPath. Specifically, the possibility to decouple notifications and data content retrievals was described. Chapter 6 described partition tolerance in SocioPath and achieving consistency between two devices by state repairs. While the required messages and their fields for these procedures were described, so far it is mostly undefined *which device sends which message when to which other device*. Different examples for this problem were discussed in Section 3.4.1 for User-Centric Networking in general and in Section 5.3 specifically for SocioPath.

In SocioPath, these decisions are delegated to the **Decision Engine (DE)**. A DE complements SocioPath to a fully functional protocol while different DEs alter SocioPath's protocol behaviour in different ways.

In this chapter, three different DEs will be presented:

- Instant-to-All
- Offload-First
- Helping-Friends

An overview on these DEs will be given in Section 7.1, before they are defined in detail in the remaining chapter.

Each DE must define its behaviour in three different workflows:

- Notification workflow
- Data objects retrieval workflow
- State repair workflow

Each workflow holds different **DE events** and **DE actions** for transferring a data object from a publishing device to the remaining devices of the closed group. Basically, a DE is defined by how it reacts on DE events and how it performs DE actions. The definitions for DE events and DE actions will be given in Section 7.2.

Since a DE is defined by its DE event reactions and DE actions, each workflow is a template for defining a DE. The three workflows are described in Sections 7.3, 7.4 and 7.5. The differences between Instant-to-All, Offload-First and Helping-Friends will be discussed for each respective workflow.

These three workflows taken together are responsible for transferring a data object from a publishing device to the remaining devices of the closed group. The overall process is shown in Section 7.6.

7.1 Overview on the Three Presented Decision Engines

This section gives an overview on the three DEs Instant-to-All, Offload-First and Helping-Friends.

7.1.1 Instant-to-All

Instant-to-All is a greedy approach for a DE, neglecting any resource conservation. After publishing a data object, the publishing device sends the notifications to as many topic owner devices as possible, as soon as possible. Each topic owner device sends a notification as soon as possible, to as many subscriber devices as possible. However, if a subscriber device is unavailable for any topic owner device, it receives the notification after the next state exchange with any topic owner device. The redundant forwarding to all topic owner devices shall achieve notification delivery as fast as possible: if topic owner devices are also often unavailable, only one topic owner device that can send the notification needs to be available for the subscriber devices.

Data objects can be piggybacked in notifications. The decision whether to piggyback is made by regarding one system-wide parameter: *sizeMax* which is the maximum allowed bytesize of a piggybacked data object. Each device which receives a notification, instantly sends a $\text{Retrieve}_{\text{Req}}$ message to all known sources unless the data object was piggybacked.

With Instant-to-All, privacy level IV is achieved, which is the highest privacy level according to the privacy model in Section 3.6: only topic owner devices deliver data objects to subscriber devices, neither the publisher nor the subscribers learn about the set of (other) subscribers.

In this chapter, the DE event decisions, as well as the DE actions are described for Instant-to-All.

Table 7.1: *Trade-offs in Instant-to-All*

	High privacy	Low delays	Resource conservation
Instant-to-All	✓	✓	✗

7.1.2 Offload-First

Offload-First is a first approach towards resource-awareness in a DE. This DE considers *Communication Cost Values (CCVs)* (see Chapter 5) of devices. A CCV is an abstract penalty for each byte the device has to send or receive – thus: the lower a device's CCV, the better. For each member of the closed group, Offload-First aims at offloading notifications to the device with the lowest CCV value first. This device becomes a *forwarding device* and has the task to deliver the notification to the remaining devices of the same user.

In contrast to Instant-to-All, the idea here is to distinguish between *stronger devices* (i.e. lower CCV) and *weaker devices* (i.e. higher CCV), to move the major traffic to the forwarding device and hence alleviate weaker devices. In addition, since the forwarding device is the only device per user which is responsible for sending notifications to the user's other devices, redundant notifications are eliminated. The lack of redundancy comes at the price of possibly high delays – if a user's strongest device is unavailable, the notifying process towards this user is stalled until the strongest device becomes available again. This applies to publisher, subscribers and topic owner.

Unlike Instant-to-All, non-piggybacked data objects do not get instantly retrieved. Instead, a system-wide CCV threshold *ccvMax* decides whether the device's communication costs are currently low enough to automatically retrieve the data object. If the current CCV exceeds the threshold, the retrieval must be triggered by an application.

Just as Instant-to-All, Offload-First achieves privacy level IV, which is the highest privacy level according to the privacy model in Section 3.6: only topic owner devices deliver data objects to subscriber devices, neither the publisher nor the subscribers learn about the set of (other) subscribers.

In this chapter, the DE event decisions as well as the DE actions are described for Offload-First.

Table 7.2: *Trade-offs in Offload-First*

	High privacy	Low delays	Resource conservation
Offload-First	✓	✗	✓

7.1.3 Helping-Friends

Helping-Friends aims at leveraging social connections between the members in a closed group. In Instant-to-All and Offload-First, devices of different users never communicate with each other, unless one of the users is the topic owner. However, the members of a closed group might not only be contacts with the topic owner, but a partly- or even fully-linked clique and hence be interested that all subscribers receive a given data object. Therefore, a subscriber device can also act as a source device for other subscribers. If at least one user has one or multiple devices with high availability and/or capability, this could compensate the weaker devices of other users.

This DE is in large parts similar to Offload-First. The main difference is that the forwarding device for notifications (i.e. the strongest device) is not determined between topic owner and one subscriber, but *per clique* between topic owner and subscribers that all are contacts of each other. Here, the strongest device of the clique becomes the forwarding device and notifies all other devices of the clique. Either the CCV (as for Offload-First) or the device's long time availability can be the main criterion here, pushing focus either towards the goal *resource conservation* or the goal *low delays*.

The cost for this approach are parts of the group members' privacy. The subscribers with strong devices learn about other subscribers, this knowledge is not exclusive to the topic owner anymore. In terms of the privacy model in Section 3.6, the privacy level is reduced from IV to III.

Note that inside a clique of contacts, all their devices can communicate which each other, since each device holds the other devices in its Devices List. Furthermore, each device can encrypt all traffic, since each device holds the necessary public key for each contact in its Contact List. A design which allows non-contacts would be more complex and is not covered here.

In this chapter, the DE event decisions as well as the DE actions are described for Helping-Friends.

Table 7.3: *Trade-offs in Helping-Friends*

	High privacy	Low delays	Resource conservation
Helping-Friends	✗	✓	✓

7.2 Device Roles, DE Events and DE Actions

Central parts of each DE workflow are the **device roles**, **DE events** and **DE actions**. Before each workflow is discussed, these parts shall be explained first, since they are substantial to all workflows.

There are four *device roles*:

- **Publishing device:** A device that publishes a data object
- **Topic owner device:** A device that belongs to the topic owner. This device can decide if the data object was published by an allowed publisher and knows which users are the topic's subscribers.

- **Remaining device:** A device that shall receive a notification about the data object and may retrieve the data object. These devices are all topic owner devices, all devices of the publisher and all devices of all subscribers
- **Source device:** A device which already holds the data object or at least parts of it (see Section 5.7.3). At least the publishing device is a source device.

During each workflow in Figure 7.7, two types of events occur: **DE events** and fix events. On DE events, the DE has to decide *how to react to that event*. It can decide whether to execute or postpone the next action. Since an action usually involves sending one or multiple messages to other devices, the respective parameters get set here (e.g. deciding whether to piggyback the data object in a notification or not). Fix events also result in actions, but the DE has no flexibility in reacting to that event. The message for the next action is always identical for every DE.

Also, during the workflow, two types of actions are required by each device: **DE actions** and fix actions. During a DE action, the DE has to decide *to which devices* a message shall be sent. For example, the DE action "notify topic owner devices" leaves it to the DE whether to notify one specific, multiple or all topic owner devices. A DE action can follow either a DE event or a fix event. A fix action must be executed by the device, but the DE cannot choose any target devices – analogous to fix events. For example, notifying an application on the same device is a fix action, no other devices are involved here.

7.3 Notification Workflow

Figure 7.2 displays the workflow for sending notifications. In the figure, DE events are blue boxes and fix events are white boxes. DE actions are blue ellipses and fix actions are white ellipses.

The notification workflow starts with the DE event **PublishFromApp** on the publishing device after a new data object is published. During this DE event, the DE has to decide whether to piggyback the data object or not.

If the publisher is not the topic owner, the publishing device has to notify one or multiple topic owner devices (DE action), since only topic owner devices hold the subscribers in their Own Topics lists.

On a topic owner device, this results in the DE event **TopicOwnerNotified**. For one or multiple topic owner devices, each of them must now decide whether to piggyback the data object (DE event) and then notify one or multiple remaining devices (DE action).

On a remaining device, this results in the fix event **GotNotify**. If the data object is piggybacked, no further steps must be taken except for the remaining device's final action to notify the application via a SODESSON notifyApp method call (see Section 4.3.3.4 for details).

7.3.1 Overview on DE events and DE actions

The DE event which requires a DE decision is as follows:

- DE event decision I (D-I): Piggyback published data object in notification?

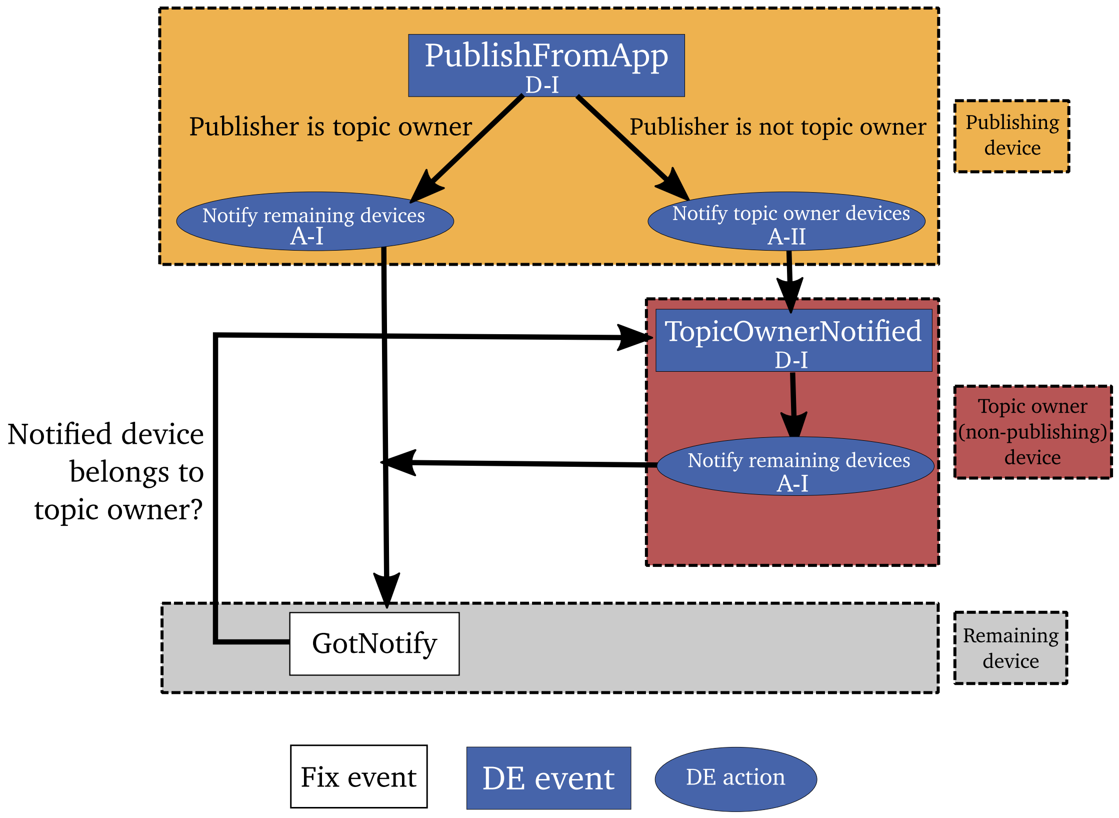


Figure 7.2: *Notification workflow*

The DE actions where the DE must decide to which devices a notification shall be sent, are as follows:

- DE action I (A-I): Notify remaining devices
- DE action II (A-II): Notify topic owner devices

Thus, for a functional notification workflow, each DE must specify its behaviour for D-I, A-I and A-II. This is now explained for the three DEs Instant-to-All, Offload-First and Helping-Friends.

7.3.2 Instant-to-All

In this section, the DE event D-I and the DE actions A-I and A-II are discussed for Instant-to-All.

D-I: Piggyback published data object in notification?

This decision is made by regarding one system-wide parameter: *sizeMax* which is the maximum allowed bytesize of the data object.

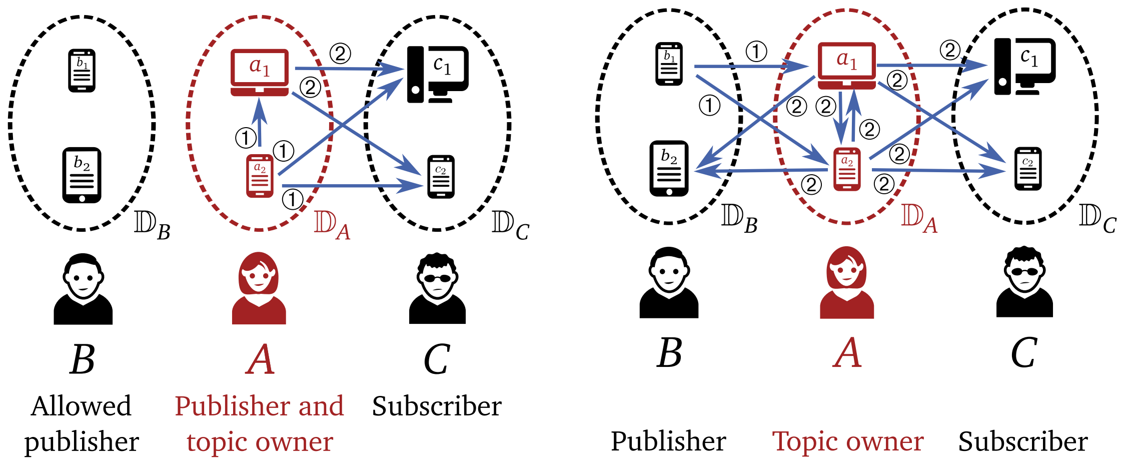
If the data object's size *objectSize* exceeds *sizeMax*, the data object is not piggybacked. Otherwise it gets piggybacked.

Since *sizeMax* is a system-wide parameter, this policy applies to the publishing device, as well as all subsequent devices that send $\text{N}\scriptstyle{\text{OTIFY}}_{\text{Req}}$ messages at some point.

A-I: Notifying remaining devices

Figure 7.3(a) displays A-I, i.e. the case where a topic owner device is the publishing device and must notify the subscriber devices. The publishing device is here a_2 of user and topic owner A. The publishing device sends a NOTIFY$_{\text{Req}}$ message to all remaining devices, directly one after another, quasi-simultaneously (step ①). Since A is the topic owner, a_2 holds the subscribers in the Own Topics list and since subscriber C is a contact of A, C's devices are in a_2's Devices List. a_2 also sends a NOTIFY$_{\text{Req}}$ message to all other topic owner devices (here: a_1) in the same manner (step ①). Note that B is only an allowed publisher, neither subscriber nor topic owner nor (here) publisher. Therefore, B's devices to not belong to the remaining devices.

If a_2 cannot reach all devices, this will be remedied by a state repair later on.



(a) Topic owner is publisher (A-I)

(b) Topic owner is not publisher (A-II then A-I)

Figure 7.3: *Instant-to-All: Notifications after publishing a new data item*

A-II: Notifying topic owner devices

Figure 7.3(b) displays A-II, i.e. the case where the publishing device is not a topic owner device. Here, the device b_1 of (allowed) publisher B, sends NOTIFY$_{\text{Req}}$ messages to all devices of the topic owner A, here a_1 and a_2 (step ①). This already closes A-II. Step ②, which is executed on each topic owner device, is again A-I: here, every topic owner simply sends a NOTIFY$_{\text{Req}}$ message to the remaining devices. There is no coordination between the topic owner devices, therefore each publisher device (except the publishing device) and each subscriber device receives as many NOTIFY$_{\text{Req}}$ messages as topic owner devices exist.

7.3.3 Offload-First

In this section, the DE event D-I and the DE actions A-I and A-II are discussed for Offload-First.

D-I: Piggyback published data object in notification?

This decision is identical to D-I in Instant-to-All (see Section 7.3.2), i.e. the decision is made by regarding one system-wide parameter: *sizeMax* which is the maximum allowed bytesize of the data object.

A-I: Notifying remaining devices

Figure 7.4(a) displays A-I, i.e. the case where a topic owner device is the publishing device. Here, a_2 of topic owner A is the publishing device. This topic owner device first checks if there is another topic owner device with a lower CCV. If yes, it forwards the notification to that device, hence triggering the DE event **TopicOwnerNotified** on that device and restarting procedure. Since each device's current CCV is sent together with the NOTIFY$_{\text{Req}}$ message, this procedure will terminate and does not loop infinitely between two devices that falsely deem the other device stronger, based on obsolete CCVs. For now, it is assumed that all devices know the correct CCV of all contacts' devices, as inferred by any recent message from these device.

The publishing device a_2 is weaker than the other topic owner device a_1. Hence, a_2 sends a NOTIFY$_{\text{Req}}$ message to a_1 (step ①). Since A is the topic owner, a_1 holds the subscribers B and C in the Own Topics list and since B and C are contacts of A, their devices are in a_1's Devices List.

a_1 now notifies the devices of subscriber B and C in different ways:

All devices of B are weaker than a_1. Hence, from a "macroeconomic" point of view (i.e. the costs for A and B taken together), it is most efficient if a_1 sends the NOTIFY$_{\text{Req}}$ to b_1 and b_2 by itself (step ②).

C on the other hand, controls the device c_1 with a lower CCV than a_1. Hence, a_1 sends a NOTIFY$_{\text{Req}}$ message to c_1 with the *forward* flag (see Section 5.4.2) set to true (step ②). By this flag, c_1 recognizes that it is asked to forward the notification to C's other devices. c_1 then forwards the notification to C's other device c_2 (step ③).

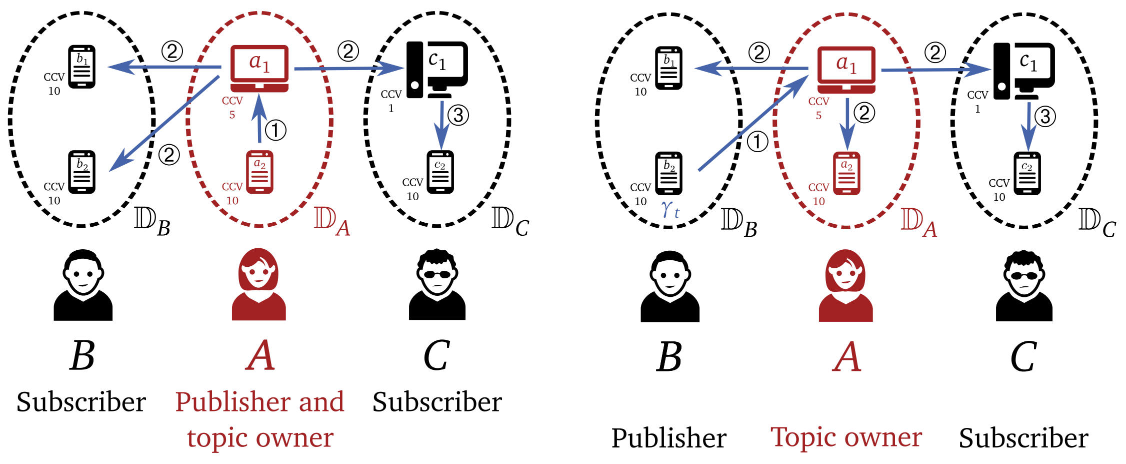
If any device cannot reach another device, it does not retry to notify the remaining devices by itself nor send the message to a fallback device. Instead, the remaining devices have to use the state repair to receive the notification later. Due to the missing redundancy compared to Instant-to-All, this can result in long delays, since the strongest device must be available.

A-II: Notifying topic owner devices

Figure 7.4(b) displays A-II, i.e. the case where the publishing device is not a topic owner device. Here, the device b_1 of (allowed) publisher B, sends a NOTIFY$_{\text{Req}}$ message to the strongest device of the topic owner A, here a_1 (step ①). This already closes A-II. The topic owner device performs the DE action A-I, i.e. Steps ② and ③ are identical as in Section 7.3.3.

7.3.4 Helping-Friends

In this section, the DE event D-I and the DE actions A-I and A-II are discussed for Helping-Friends. This DE requires additional fields in the NOTIFY$_{\text{Req}}$ messages that are not provided by SocioPath itself. Such extensions for individual Decision Engines are not in the current design, however a



(a) Topic owner is publisher **(b)** Topic owner is not publisher

Figure 7.4: *Offload-First: Notifications after publishing a new data item*

generic field that a DE can fill with its individual extension data structure is one possible solution here. These extensions are discussed after the workflow description.

D-I: Piggyback published data object in notification?

This decision is identical to D-I in Instant-to-All (see Section 7.3.2), i.e. the decision is made by regarding one system-wide parameter: *sizeMax* which is the maximum allowed bytesize of the data object.

A-I: Notifying remaining devices

Figure 7.5(a) displays A-I, i.e. the case where a topic owner device is the publishing device, here a_2 of user and topic owner A. B, C and D are subscribers, i.e. they are all contacts of A. Furthermore, C and D are contacts of each other. It is assumed that all devices know the correct CCV of all contacts' devices.

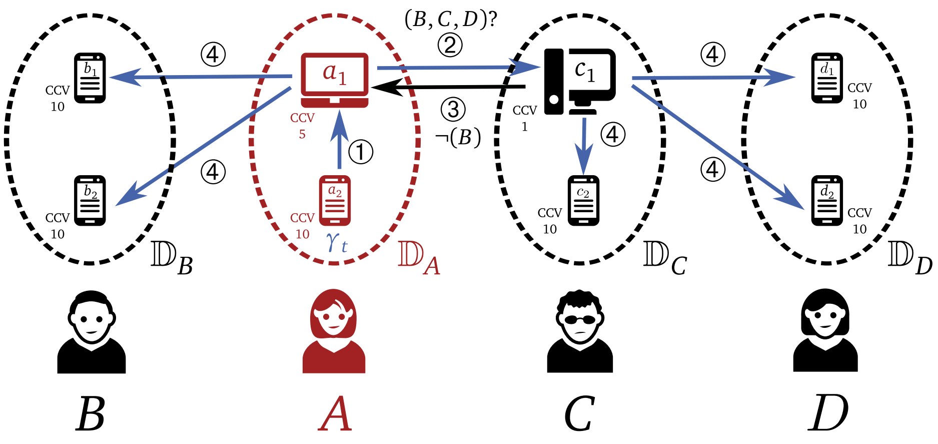
The publishing device a_2 is weaker than the other topic owner device a_1. Hence, a_2 sends a NOTIFY$_{\text{Req}}$ message to a_1 (step ①) for disseminating it to the subscriber devices. So far, this step is identical to the example in Offload-First.

a_1 now checks in its Devices List if it knows a subscriber device which is stronger than a_1 itself. c_1 of subscriber C is stronger than a_1. a_1 sends a NOTIFY$_{\text{Req}}$ message to c_1 *including the list of all subscribers* (B, C and D) in the field *forwardToUsers*, and *forward* set to true (step ②).

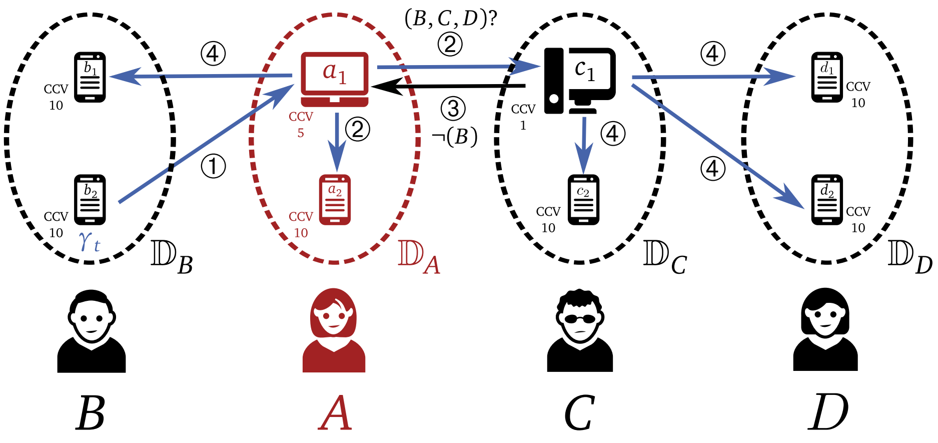
c_1 now checks in its Contact List if it does not know any of the given subscribers (here B) and returns that set in the *declinedForwards* field of the NOTIFY$_{\text{Rsp}}$ message (step ③).

a_1 now repeats the process for the remaining subscribers, until no *declinedForwards* are left. Here, B is the remaining subscriber. a_1 checks for the remaining subscriber devices if there is a subscriber device which is stronger than a_1. This is not the case, since B's devices are all weaker than a_1. Hence, a_1 sends the notifications to B's devices by itself, analogously to Offload-First (step ④).

In the meantime, c_1 adds the notification into a *forwarding map* which holds a to-be-forwarded notification together with the devices c_1 has commited itself to forwarding (here: c_2, d_1, d_2). c_1 sends a NOTIFY$_{\text{Req}}$ message to each of these devices (step ④). As soon as c_1 a NOTIFY$_{\text{Rsp}}$ message, the respective combination of notification and device is deleted from the forwarding map. If entries remain, because a device is unavailable, this is resolved during state repair.



(a) Topic owner is publisher



(b) Topic owner is not publisher

Figure 7.5: *Helping Friends*

A-II: Notifying topic owner devices

Figure 7.5(b) displays A-II, i.e. the case where the publishing device is not a topic owner device. Here, the device b_1 of (allowed) publisher B, sends a NOTIFY$_{\text{Req}}$ message to the strongest device of the topic owner A, here a_1 (step ①). This already closes A-II. The topic owner device performs the DE action A-I, i.e. Steps ② to ④ are identical as above.

Extensions

Regardless of any DE, only the topic owner devices know about a topic's subscribers, as given by their Own Topics lists. Thus, if one subscriber shall help another subscriber, the topic owner device must tell the helping subscriber device which other users are also subscribers. This is achieved by adding the *forwardToUsers* field in the $\text{NOTIFY}_{\text{Req}}$ message (see Table 7.4). In combination with the already existing boolean forward flag, the sending topic owner device can ask the receiving device to help deliver notifications to the device of the users in the *forwardToUsers* field.

Likewise, the subscriber device which was asked to help is able to decline forward requests. The helping subscriber might not be contacts with all users inside the received *forwardToUsers* field. The topic owner is not aware of this before sending the $\text{NOTIFY}_{\text{Req}}$ message, since the topic owner does not know which users are contacts of the helping subscriber's. Therefore, the *declinedForwards* field is added to $\text{NOTIFY}_{\text{Rsp}}$ message (see Table 7.5). The helping subscriber device can enter the declined subset of the received *forwardToUsers* field into the *declinedForwards* of the $\text{NOTIFY}_{\text{Rsp}}$ message.

Note that by evaluating the differences between sent *forwardToUsers* fields and returned *declinedForwards* fields, a topic owner can learn about the helping subscriber's contacts. This is also a loss of privacy for the helping subscriber.

Table 7.4: *Additional fields inside a* NOTIFY_{Req} *message. See Table 5.16 for the entries omitted here*

Data type	Field name	Description
...	...	...
`UID`	publisherUserId	User ID of the data object's publisher. Used to identify the original publisher, since this field can differ from *senderUserId*.
`Boolean`	forward	This flag indicates to the Decision Engine whether the receiving device is asked to forward the $\text{NOTIFY}_{\text{Req}}$ message to other devices.
`Array<UID>`	**forwardToUsers**	*(Optional:)* Can be used in combination with the *forward* flag to ask a device to forward to other users.

Table 7.5: *Additional fields inside a* NOTIFY_{Rsp} *message.*

Data type	Field name	Description
`Boolean`	ok	General purpose for ACK / NACK
`Array<UID>`	**declinedForwards**	*(Optional:)* Used to decline a subset from the $\text{NOTIFY}_{\text{Req}}$ message's **forwardToUsers** field

7.4 Data Object Retrieval Workflow

As soon as a device was notified, the workflow for data object retrieval becomes relevant next. It is displayed in Figure 7.6.

In case the data object was not piggybacked, it has to be retrieved by the notified device. This is indicated by the DE event RetrieveRequired which lets the device's DE decide if and when

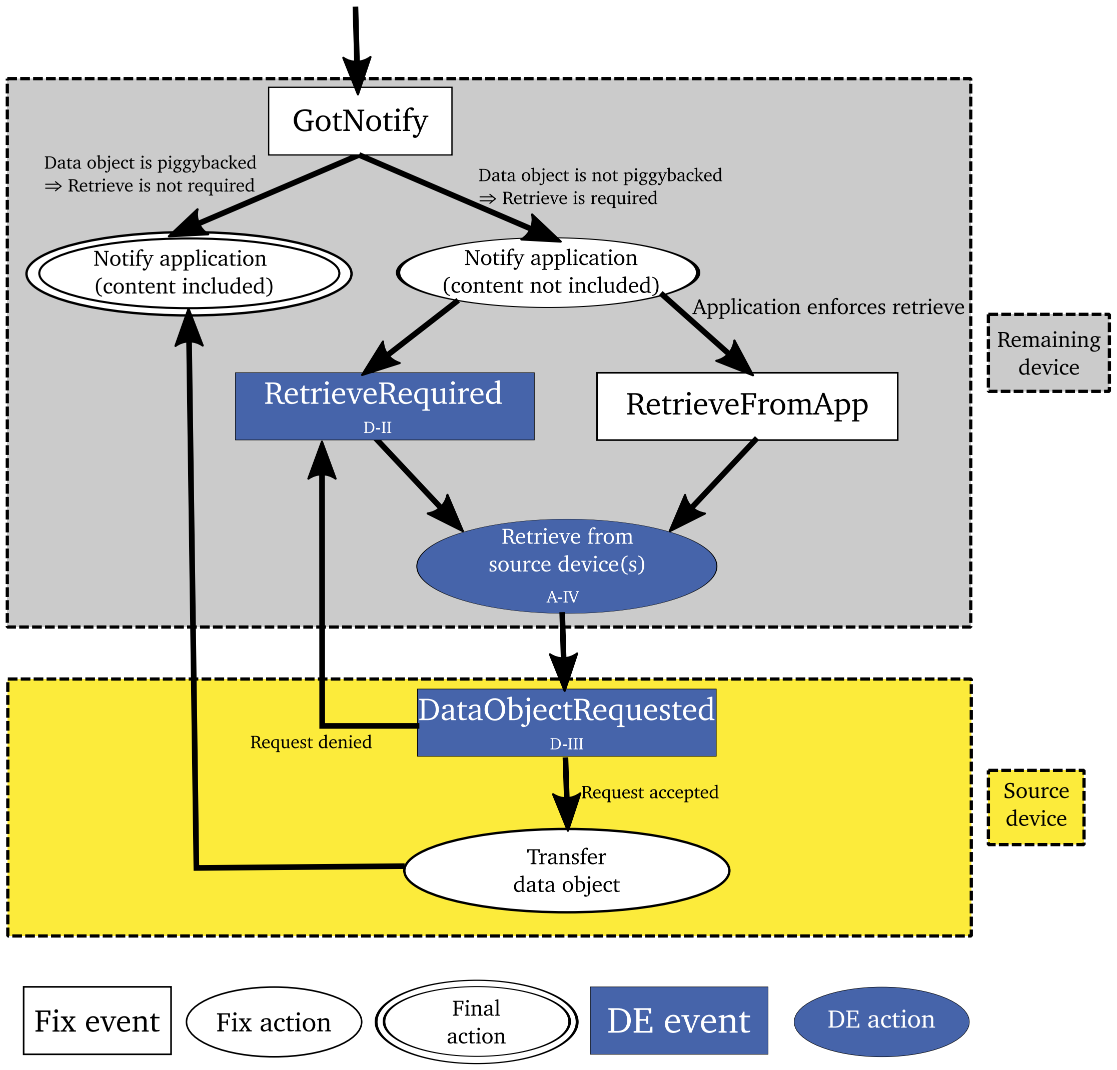

GotNotify
Data object is piggybacked ⇒ Retrieve is not required
Data object is not piggybacked ⇒ Retrieve is required
Notify application (content included)
Notify application (content not included)
Application enforces retrieve
Remaining device
RetrieveRequired D-II
RetrieveFromApp
Retrieve from source device(s) A-IV
DataObjectRequested D-III
Request denied
Request accepted
Source device
Transfer data object
Fix event
Fix action
Final action
DE event
DE action

Figure 7.6: *Data object retrieval workflow*

to retrieve the data object. Additionally, the application is notified via a SODESSON notifyApp method call where the *content* field is empty and the *inline* field is set to false (see Section 4.3.6). The application can react to this call and hence enforce the retrieval of the data object, overriding any decision the DE might have made in the context of **RetrieveRequired**. This case results in the fix event **RetrieveFromApp** – the data object must be retrieved and the DE cannot decide here.

In order to initiate data object retrieval, the device has to send RETRIEVE$_{\text{Req}}$ message from one or multiple source devices (DE action).

As soon as a source device receives the RETRIEVE$_{\text{Req}}$ message, the DE event **DataObjectRequested** is triggered. Here, the source device must decide whether to accept or decline the request. The source device tells the requesting subscriber in the corresponding RETRIEVE$_{\text{Rsp}}$ message. If the request is accepted, the data object transfer has to be performed. If the request is denied, the requesting returns back to the **RetrieveRequired**. On completion, the remaining device's final action is performed: notifying the application via a SODESSON notifyApp method call with a non-empty *content* field.

7.4.1 Overview on DE events and DE actions

The DE events which require a DE decision are as follows:

- DE event decision II (D-II): Send retrieval request?
- DE event decision III (D-III): Accept retrieval request?

The DE actions where the DE must decide from which devices it shall retrieve, are as follows:

- DE action III (A-III): Retrieve from source devices

7.4.2 Instant-to-All

D-II: Send retrieval request?

Instant-to-All always instantly triggers the retrieval of a data object from all known sources, as soon as the device receives a $\text{NOTIFY}_{\text{Req}}$ message without piggybacked data object. The DE tries this once for every received $\text{NOTIFY}_{\text{Req}}$ message. If needed, the application must initiate repeatedly the data object retrieval by triggering the fix event **RetrieveFromApp**.

D-III: Accept retrieval request?

In Instant-to-All, retrieval requests are always accepted if the device which receives the $\text{RETRIEVE}_{\text{Req}}$ message is actually a source device.

A-III: Retrieving from source devices

As soon as either the fix event **RetrieveFromApp** occurs or the decision D-II for DE event **RetrieveRequired** results in a retrieval request, the device sends a $\text{RETRIEVE}_{\text{Req}}$ message to all known source devices.

7.4.3 Offload-First and Helping-Friends

Offload-First and Helping-Friends behave identically for D-II, D-III and A-III and are thus regarded together in one section.

D-II: Send retrieval request?

For triggering the retrieval of a data object automatically, Offload-First takes CCVs into account: As soon as device is notified, the event **RetrieveRequired** is triggered. If the device has a lower CCV than a threshold *ccvMax*, it sends a $\text{RETRIEVE}_{\text{Req}}$ message to all known source devices. Otherwise, the DE does nothing and the application must initiate the retrieval. Note: a simple improvement here would be to recheck for pending retrievals as soon as the CCV drops below *ccvMax*, for example if a mobile device switches from a metered, mobile connection to a free WiFi.

D-III: Accept retrieval request?

As soon as a device receives a $\text{RETRIEVE}_{\text{Req}}$ message, the DE event **DataObjectRequested** is triggered. In Offload-First, a source device accepts a retrieval request, if it has a lower CCV than a threshold *ccvMax*, otherwise it declines the request.

A-III: Retrieving from source devices

As soon as either the fix event **RetrieveFromApp** occurs or the decision D-II for DE event **RetrieveRequired** results in a retrieval request, the device sends a $\text{RETRIEVE}_{\text{Req}}$ message to all known source devices.

7.5 State Repair Workflow

The workflow for state repairs is simple and consists of only one DE event and one DE action

- DE event decision IV (D-IV): Initiate state repair now?
- DE action IV (A-IV): Send $\text{STATE}_{\text{Req}}$ to other devices

The DE runs periodically (for example every second) into the DE event **SendState**. Here, the DE has to decide (D-IV) whether to actually initiate state repair or ignore the event. For example, a possible decision could be "initiate state repair every 5 minutes". In that case, the DE would ignore the DE event **SendState** 299 times and send a $\text{STATE}_{\text{Req}}$ to one or multiple source devices (DE action A-IV).

7.5.1 Instant-to-All

D-IV: Initiate state repair now?

A device sends $\text{STATE}_{\text{Req}}$ messages within a so-called *state round*. Hence, it has to decide when to start a new state round. There are two possible triggers for a new state round:

- Proactively: A new state round is started after $n :=$ randUniform($\{0.5\cdot$ *statePeriod*, $\ldots, 1.5\cdot$ *statePeriod*$\}$) seconds, with n being drawn after the previous state round. *statePeriod* is a system-wide parameter. The randomness is used in order to prevent oscillation effects between multiple devices.
- Reactively: A new state round is started as soon as an unavailable device becomes available again. This requires the device to have a mechanism on operating system level to detect whether it is available or unavailable for other devices (e.g. by having Internet access).

A-IV: Send $\text{STATE}_{\text{Req}}$ to other devices

A state round on a device a_1 is performed as follows: For each contact X, X's devices are collected from the Devices List in arbitrary order. Again for each contact X, a_1 generates a $\text{STATE}_{\text{Req}}$ message, based on the the relevant topics $\mathbb{R}_{A,X}$ (see Section 6.1.1). a_1 sends this $\textsc{State}_{\text{Req}}$ message to the first device x_1 of that contact. Hence, for n contacts, a_1 sends n $\textsc{State}_{\text{Req}}$ messages in one burst. As soon as a state repair is resolved with one of these devices, a_1 creates a new $\textsc{State}_{\text{Req}}$ message based on the relevant topics for that contact. a_1 sends this new $\textsc{State}_{\text{Req}}$ message to the next device x_2 of that contact. This procedure continues for all devices of all contacts.

Hence, in a state round different contacts are handled parallelly, while the devices of the same contact are handled subsequently. The state repair from user A to different contacts X and Y are based on different sets of relevant topics $\mathbb{R}_{A,X}$ and $\mathbb{R}_{A,Y}$, which allows for parallelism. The subsequent handling of X's devices prevents a_1 from unnecessarily receiving notification multiple times. Assume that x_1's and x_2's Data Storages hold an entry about a data object γ_t that a_1 has missed. After the state repair with x_1, a_1 updates its Data Storage first before sending a $\textsc{State}_{\text{Req}}$ message to x_2. This way, x_2 does not resend a notification about γ_t. If a_1 would have sent the $\textsc{State}_{\text{Req}}$ message to x_1 and x_2 at the same time, it would have unnecessarily received the notifications twice.

7.5.2 Offload-First

D-IV: Initiate state repair now?

This decision is identical to D-IV in Instant-to-All (see Section 7.5.1), i.e. state rounds are initiated either proactively or reactively.

A-IV: Send $\textsc{State}_{\text{Req}}$ to other devices

Analogous to Instant-to-All, a device sends $\textsc{State}_{\text{Req}}$ messages within a so-called *state round*, triggered either periodically or reactively (see Section 7.5.1). Within one state round, each contact's devices are handled subsequently. This means, as soon as device a_1 has resolved a state repair with device x_1 of contact X, a_1 sends a $\textsc{State}_{\text{Req}}$ message to another X'x device x_2, for all devices of X.

Unlike Instant-to-All (which chooses a random device order), Offload-First takes CCVs into account, consequently X's devices are ordered by their CCVs, strongest device first. The consideration here is that if stronger devices are asked first, they can already resend most of the missed notifications. Weaker devices only have to resolve the remaining notifications (if any exist) that the stronger devices did not know about.

7.5.3 Helping-Friends

The DE event decisions D-I to D-IV in Helping-Friends are identical to those Offload-First. See Sections 7.3.3 to 7.5.2.

A-IV: Synchronize State with other devices

For the device x of user X, which sends the $\textsc{State}_{\text{Req}}$ message, A-IV in Helping-Friends is almost identical to A-IV in Offload-First. However, before the actual state round begins, x sends the first $\textsc{State}_{\text{Req}}$ message to the strongest device s of all of X's contacts. It is assumed that if a device holds notifications in its forwarding map for x, it is probably s. Hence, s can already send missed notifications to x before the actual state round begins and alleviate other devices from resends.

Generally, if a device y receives a $\text{STATE}_{\text{Req}}$ message from x, it checks in its forwarding map if it contains notifications for x, i.e. notifications y has committed to forwarding to x but has not succeeded yet. If true, y increases the number in the *numMisses* field of the $\text{STATE}_{\text{Rsp}}$ message accordingly and sends these notifications as missed notifications besides the actual STATE repair about the relevant topics between X and Y.

Note: If a notification from the forwarding map has another topic owner than X or Y, y cannot recognize whether x has already received said notification from another device. A $\text{STATE}_{\text{Req}}$ message only contains information about the relevant topics between X and Y. In that case, it might happen that x receives the same notification multiple times.

7.6 Overall Process

So far, Decision Engines were described in three separate workflows: notification, data object retrieval and state repair. The former two workflows are put together in Figure 7.7. State repair is not shown in the figure, since it is a maintenance task, that can run periodically and is not triggered by events such as incoming notification or publishing of a new data object.

Table 7.6 summarizes the connections between device role, decisions on DE events and DE actions.

Table 7.6: *Summary of device roles, decisions on DE events and DE actions*

DE Event	Device Role	Triggered when	Decision	Action
PublishFromApp	Publishing device (from topic owner)	Application has published a new data object	D-I	A-I
PublishFromApp	Publishing device (not from topic owner)	Application has published a new data object	D-I	A-II
TopicOwnerNotified	Topic owner device	Topic owner's device has received a $\text{NOTIFY}_{\text{Req}}$ message from another user's device	D-I	A-I
RetrieveRequired	Remaining device	Device has received a $\text{NOTIFY}_{\text{Req}}$ message where the data object was not piggybacked	D-II	A-III
DataObjectRequested	Source device	Device has received a $\text{RETRIEVE}_{\text{Req}}$ message	D-III	Transfer
SendState	Any device	to be defined by DE	D-IV	A-IV

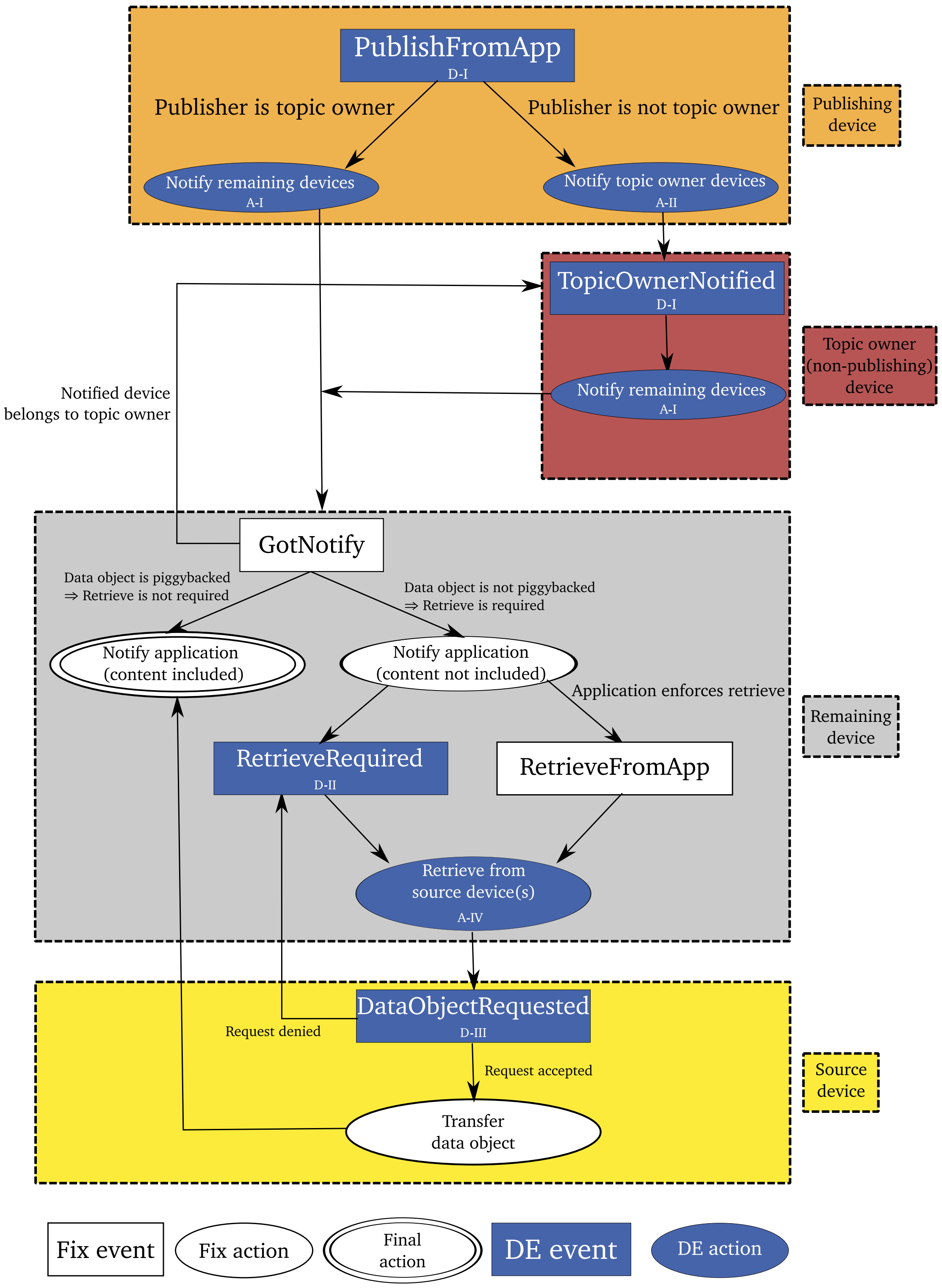

PublishFromApp
D-I
Publisher is topic owner
Publisher is not topic owner
Notify remaining devices
A-I
Notify topic owner devices
A-II
Publishing device
TopicOwnerNotified
D-I
Notify remaining devices
A-I
Topic owner (non-publishing) device
Notified device belongs to topic owner
GotNotify
Data object is piggybacked ⇒ Retrieve is not required
Data object is not piggybacked ⇒ Retrieve is required
Notify application (content included)
Notify application (content not included)
Application enforces retrieve
RetrieveRequired
D-II
RetrieveFromApp
Retrieve from source device(s)
A-IV
Remaining device
DataObjectRequested
D-III
Request denied
Request accepted
Transfer data object
Source device
Fix event
Fix action
Final action
DE event
DE action

Figure 7.7: *Decision Engine workflow*

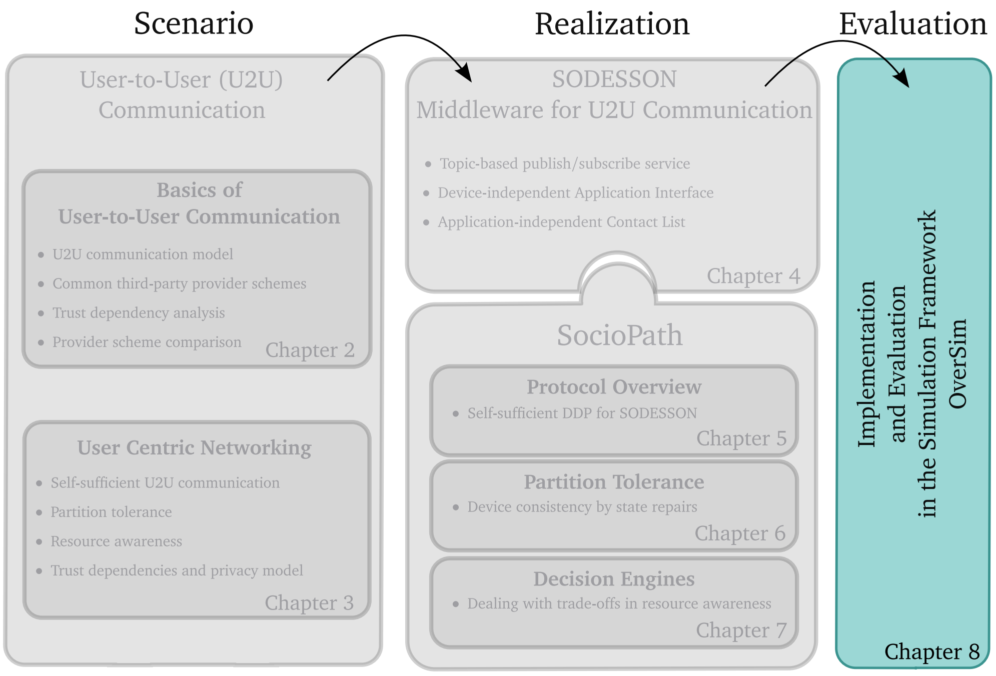
7.7 Conclusion

SocioPath is designed to be resource-aware by supporting different forwarding policies for data objects. Depending on different factors, one behaviour can be more preferrable than another. Such factors are for example the distribution of personal devices with regard to number, device availabilites and device capabilities) or different demands on privacy. In order to enable different forwarding policies, SocioPath defines different points in its protocol cycle which are left to be defined by a so-called Decision Engine (DE) – so-called DE events and DE actions: For each DE event, a DE must define to react to that event. Each DE action involves sending messages to other devices – here the DE must select a suitable subset of the targeted devices.

DE events and DE actions are a substantial part of three different workflows: notifications, data object retrieval and state repairs. Each Decision Engine takes this workflow as a template, and specifies its behaviour on these well-defined events and action. This was presented for three example DEs: Instant-to-All, Offload-First and Helping-Friends, where each of them aims for a different trade-off between privacy, low delays and resource conservation.

Chapter 8

Evaluation



Scenario
Realization
Evaluation
User-to-User (U2U) Communication
Basics of User-to-User Communication
U2U communication model
Common third-party provider schemes
Trust dependency analysis
Provider scheme comparison
Chapter 2
User Centric Networking
Self-sufficient U2U communication
Partition tolerance
Resource awareness
Trust dependencies and privacy model
Chapter 3
SODESSON Middleware for U2U Communication
Topic-based publish/subscribe service
Device-independent Application Interface
Application-independent Contact List
Chapter 4
SocioPath
Protocol Overview
Self-sufficient DDP for SODESSON
Chapter 5
Partition Tolerance
Device consistency by state repairs
Chapter 6
Decision Engines
Dealing with trade-offs in resource awareness
Chapter 7
Implementation and Evaluation in the Simulation Framework OverSim
Chapter 8

Figure 8.1: *Placement of Chapter 8 in the big picture*

8.1 The Overlay Simulation Framework *OverSim*

For evaluating the performance of SocioPath, a prototype was implemented for the overlay simulation framework *OverSim* [7] [8]. OverSim enables implementations of new network protocols which can then be evaluated with a configurable number of devices in a simulated network. The following helpful features simplify the implementation:

- **Device availability models:** each device is bound to a *churn generator* which sets the device into subsequent availability and unavailability phases according to a probabilistic model, e.g. Weibull or Pareto. The parameters of each probability distribution are configurable.
- **Realistic network latencies:** Latencies for sent messages between two devices are based on the Internet-realistic Skitter [57] dataset of the CAIDA project [17].
- **Statistics:** During simulation, relevant protocol data can be collected for each device. At the end of the simulation, these can be either collected as single values or be used to calculate a single minimum / mean / maximum value over all devices.
- **Graphical interface:** a GUI allows for protocol debugging, e.g. by inspecting device states and message contents

8.1.1 Own Contributions to *OverSim*

Figure 8.2 shows a UML class diagram of the essential modules that are relevant for the evaluation of SODESSON and SocioPath. Modules in grey were already provided by OverSim, all colored modules are own contributions. The colors of the modules match the colors from Chapter 4 which described the SODESSON middleware: applications are yellow, SODESSON is blue and modules related to SODESSON's DDP (i.e. SocioPath) are green. Additionally, there are two new global modules for OverSim in red.

The tasks of the newly developed modules are as follows:

- **GlobalUserObserver** stores the mapping between each device and its user. Additionally, it stores a global social graph (i.e. the information which user has a social relationship with whom). Note that two users in a social relationship are not contacts in SODESSON until their devices have established them as contacts. The latter is done via the *SodessonContactMaker* application. Furthermore, this module offers an abstraction for discovering devices to *SocioPath*. It is used by *SocioPath* when a user adds a new device (see Section 5.6.2) or adds a new contact (see Section 5.6.3). The *GlobalUserObserver* provides the sending device automatically with an IP address of the target device where the sending device can send an $\text{BOOTSTRAP}_{\text{Req}}$ or $\text{NEWCONTACT}_{\text{Req}}$ message respectively. In any real world implementations, this would require extra an mechanism such as a name service, discovery messages broadcasts, etc.
- **SodessonContactMaker** is an application which does not publish data objects, but instead only calls the `addContact()` method. The application learns from the module *GlobalUser-Observer* the social neighbors of the user which runs the application. From these social neighbors, one user is randomly picked and used for the `addContact()` call. This simulates the decision of two friends (social neighbors) to become SODESSON contacts (call of `addContact()`). This is repeated over and over until all social connections of a user are eventually SODESSON contacts.
- **GlobalSodessonTestObserver** tracks all information to each device's SodessonTestApp and helps to abstract from details that would be normally handled by the application. If a

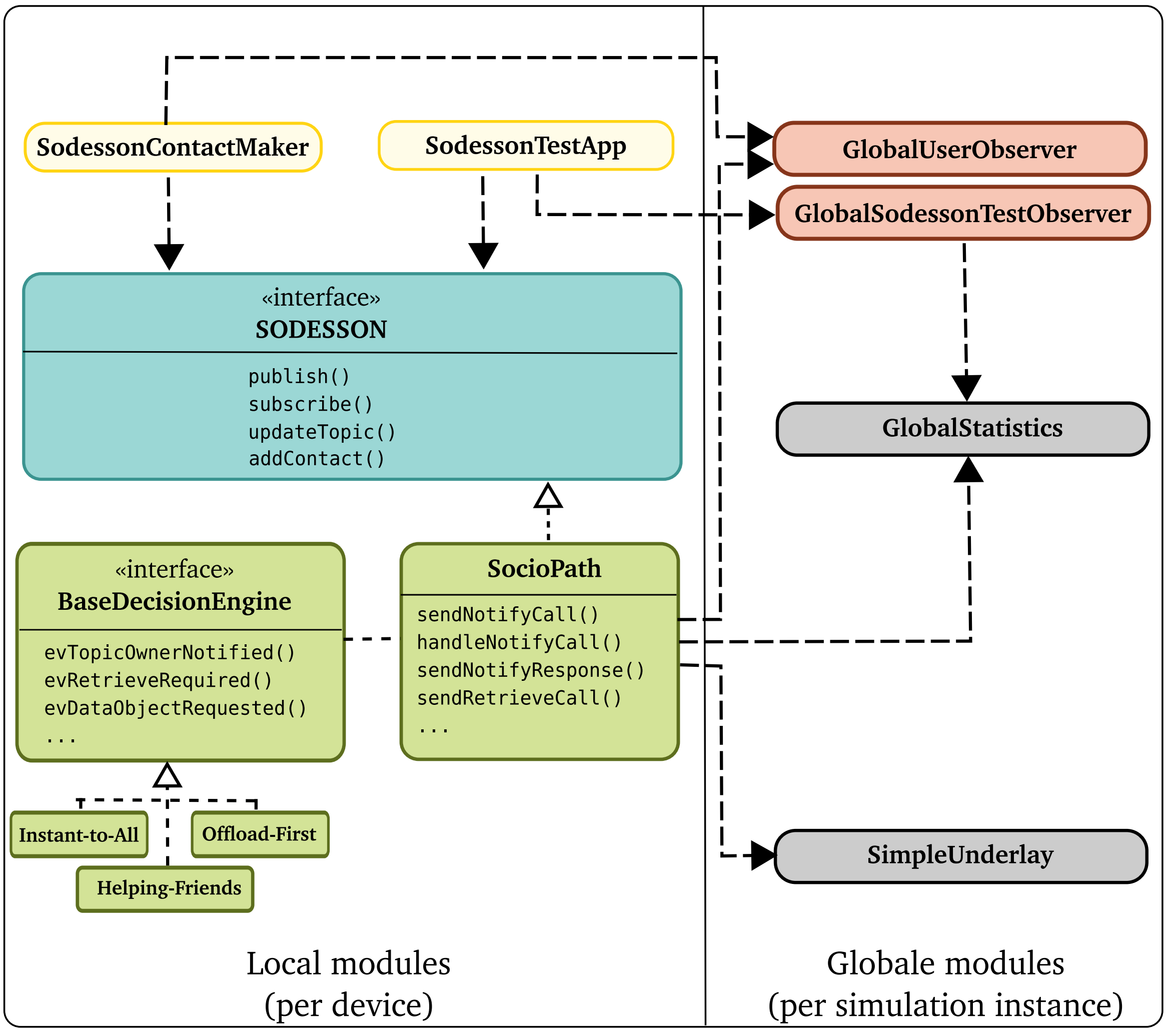


Figure 8.2: *Implementation of SODESSON and SocioPath in OverSim. Grey modules were provided by OverSim, colored modules are own contributions.*

new topic gets generated on the SodessonTestApp on device a, the topic is stored in the GlobalSodessonTestObserver. If the SodessonTestApp on device b subscribes to a topic, it selects an existing topic that was previously stored in the GlobalSodessonTestObserver. This subscription is also stored in the GlobalSodessonTestObserver. If device a publishes a new data object for a given topic, the GlobalSodessonTestObserver knows which devices should receive that data object and tracks the individual delivery delays. At the end of the simulation, these delivery delays are used for the statistics.

- **SodessonTestApp** simulates an application for U2U communication and the respective user behaviour. It runs on each simulated device and – according to a given configuration – generates new topics, subscribes to topics, publishes new data objects and accepts received data objects.

- **SODESSON** provides methods to the applications as described in Chapter 4. For this proof-of-concept in OverSim, SODESSON is designed as an interface which is implemented by the SocioPath class.
- **SocioPath** implements the *SODESSON* interface and creates the messages as described in Chapters 5 and 6. Additionally, it holds the SODESSON module Local Data Storage (see Section 4.3.5) as a data structure, since SODESSON is merely an interface in this implementation. The methods in this module after either triggered by an application via the SODESSON interface, by a received message sent by another device or by the respective Decision Engine.
- **BaseDecisionEngine** is an interface which is implemented by each Decision Engine. There is a mutual dependency between SocioPath and the Decision Engine. For example, if SocioPath receives a NOTIFY$_{\text{Req}}$ message where the RetrieveRequired flag is set (see Section 5.7), it calls the `evRetrieveRequired()` event method from the BaseDecisionEngine interface. Vice-versa, if a Decision Engine for example decides to retrieve a object, it calls `sendRetrieveCall()` method in *SocioPath*.
- **Instant-to-All / Offload-First / Helping-Friends** are implementations of the Decision Engines described in Chapter 7. They implement the *BaseDecisionEngine* interface.

Of the existing modules that are provided by OverSim, the following are relevant for the interaction with the new modules:

- **GlobalStatistics**: used by *GlobalSodessonTestObserver* and *SocioPath*. The *GlobalSodessonTestObserver* sends information about successful deliveries (and their delays). SocioPath sends information about the number of sent messages, their size and the respective costs. Both statistics (delays and costs) are performance metrics in this chapter.
- **SimpleUnderlay**: offers a UDP/IP interface to SocioPath, which lets SocioPath send arbitrary messages to the IP address of target device. On sending a messaging, SimpleUnderlay adds an Internet-realistic latency based on network coordinates before the message arrives at the target device [53].

8.1.2 Simplifications

Each *SocioPath* message is completely sent via *SimpleUnderlay* in a single packet, regardless of its size. Since OverSim offers a UDP/IP interface, technical restrictions of UDP like maximum transmission units are abstracted here. Instead only the SocioPath message sizes are regarded (i.e. the transport protocol's payload), independent of any underlying transport protocol.

8.2 Simulation of User-Centric Networks

OverSim simulates UCNs with the help of four input parameter classes:

- Social graph

- Device classes
- Device ownerships
- Applications

Each simulation run generates a configurable number of user and devices entities. Between some users, social relationships get created, i.e. a *social graph* is generated. The devices are of different *device classes* with different availabilities and capabilities. Each generated device gets mapped to exactly one user, thus creating *device ownerships*. One user of the social graph becomes the topic owner (TO) while his contacts use *applications* on their owned devices. Applications subscribe to a given topic, publish data objects or accept data objects that were published by the same application on another device.

These four parameter classes are now discussed in detail.

8.2.1 Social Graph

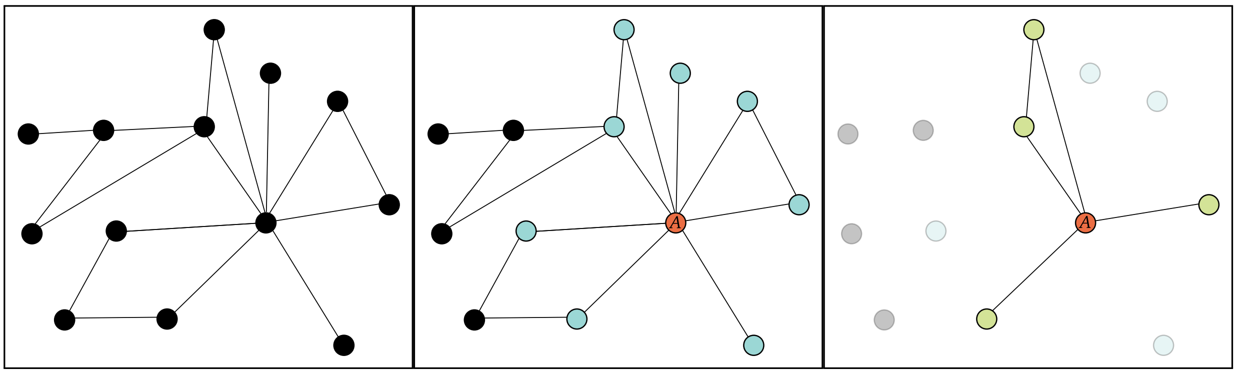
The basis for a simulated social graph is a real-world data set from Facebook, generated and made available by Viswanath et al. [114]. The data set consists of anonymized, bidirectional links that were inferred by crawling Facebook's New Orleans regional network. The statistics of the data set were analyzed for this thesis with the help of OverSim's statistic generation and the graph analyzer tool *Gephi* [5]. They are presented in Table 8.1. The found numbers differ slightly from the numbers as published by the authors.

Table 8.1: *Statistics of the Facebook New Orleans data set [114]*

Overall number of users	63,731
Overall number of bidirectional contact links	1,545,686
Number of links per user: Average	48.5
Number of links per user: Standard deviation	75.8
Number of links per user: Median	20
Number of links per user: Minimum	1
Number of links per user: Maximum	2113
Average clustering coefficient:	0.253

Social neighborhood generation

In order to be able to perform realistic simulation on a packet level, the number of simulated devices and thus users has to be limited. Only a subset of the New Orleans data set is regarded per simulation run. A random sub-graph is generated from this data set, i.e. the social neighborhood of a so-called **anchor user**. This anchor user is randomly chosen, together with a configurable number of his social relationships. In turn, any links between these social relationship are also kept intact. This sub-graph is stored in the *GlobalUserObserver* and is the basis for SODESSON contacts that get established during the simulation run.



(a) Overall social graph

(b) Random selection of a user with at least $(n_{Users} - 1) = 4$ links as anchor user. *A* with 8 links fulfills this requirement.

(c) Random selection of $(n_{Users} - 1) = 4$ neighbors of the anchor user, keeping links between them intact.

Figure 8.3: *Social graph generation for* $n_{Users} = 5$

In the simulation runs, the anchor user will be the topic owner. Since in a UCN, all users are contacts of the topic owner, this sub-graph is used as a base set for selecting the users in each UCN.

The sub-graph generation is displayed in Figure 8.3. Starting point is the overall data set (schematized as a social graph in Figure 8.3a). Before each simulation run, an integer parameter n_{Users} defines how many users are to be simulated. Then, anchor user *A* with at least $(n_{Users} - 1)$ links, i.e. social relationships, is randomly picked from the dataset (Figure 8.3b). Then, $(n_{Users} - 1)$ from all of *A*'s social relationships are randomly picked (Figure 8.3c). In turn, all links (if any) between these social relationship are also kept intact.

The result is a subset of *A*'s 1-hop social neighborhood, reduced to $(n_{Users} - 1)$ relationships. As will be discussed below, application use-cases revolve around the anchor user, with him being the TO.

From Table 8.1 it can be seen that the largest sub-graph for a simulation is 2114 users, i.e. the user with the maximum number of links as anchor user and his 2113 links.

As the default for the simulations, 20 users will be simulated, i.e. the anchor user with 19 links. This is close to the measured median of the dataset (20 links).

8.2.2 Device Classes

Three different device classes are simulated:

- *Smartphone:* High availability, low capability
- *Laptop:* Low availability, medium capability
- *Home Server:* High availability, high capability

These device classes were already presented as examples in Section 3.2.3. They differ in two aspects: *availability* and *capability*. For a definition of availability and capability, see Section 3.2. These two aspects are modeled as follows.

Availability model

The different device classes have different availabilities. Each device's availability can be modeled by alternating **lifetime** and **deadtime** phases, as proposed in [118]. If and only if two devices are in a lifetime phase (or **alive**) at the same time, they are available for each other and can communicate with each other during that time. If a device is in a deadtime phase, it is called **dead** until it becomes alive again.

[73] proposes a synthetic approach to modeling churn and communication delays in so-called 1-hop social overlays. Like [118], the model also follows the notion of subsequent lifetime and deadtime phases. The length of these phases follow a heavy-tailed, pareto-shifted distribution with shape parameter $\alpha = 3$. This is also applied for the simulations here.

For lifetime phases, the authors use an expected value of 30 minutes, whereas for deadtime phases an expected value of 1 hour is used. These numbers are based on measurements in P2P filesharing network. They are used as the default values for the *Laptop* device class. This is based on the assumption that a typical file sharing client is running on a computer and is sitting in the background while a user uses his computer, hence the connectivity of the filesharing client being equal to the connectivity of the laptop. Thus, a laptop has an average availability ratio of 1/3.[1]

For the smartphone device class (high availability), an expected lifetime phase length of 16 hours is assumed in this evaluation. This reflects a continuous availability during a full working day plus evening. At night, a smartphone is assumed to be switched off. This is modeled by an expected deadtime phase length of 8 hours. The average availability ratio of a smartphone is therefore 2/3.

Home servers are assumed to have no deadtime phases at all, thus a server has an availability ratio of 1.

The discussed parameters are summarized in Table 8.2.

Table 8.2: *Parameter values for three device classes*

Device class	Expected lifetime phase length	Expected deadtime phase length	CCV
Smartphone	16 hours	8 hours	10
Laptop	30 minutes	1 hour	5
Home Server	∞ hours	0 seconds	1

Laptops and smartphones differ in another important aspect besides their average availability: the expected lengths of lifetimes and deadtimes are much longer for smartphones than for laptops (16 hours vs. 30 minutes and 8 hours vs. 1 hour respectively). This means that once a smartphone is dead it will be probably dead for a long time, unlike a laptop.

[1]The average availability of device class C is: $\bar{a}_C := \frac{E(L_C)}{E(L_C)+E(D_C)}$ with $E(L_C)$ being the expected length of a lifetime phase for devices of class C and $E(D_C)$ being the expected length of a deadtime phase respectively.

Figure 8.4 illustrates this aspect. Here, the pareto-distributed lifetimes and deadtimes of smartphones and laptops are displayed as cumulative distribution functions (CDFs). A CDF is defined as $F_X(x) := P(X \leq x)$. It describes the probability P that a random variable X (here: length of a lifetime / deadtime) will be less than or equal to x. For a pareto-shifted distribution with shape parameter $\alpha = 3$, it is $F_X(x) = 1 - \left(\frac{2}{3} * \frac{E(X)}{x}\right)^3$ with $E(X)$ being the expected value for X. For example, about 87% of all smartphones deadtimes are shorter than 38400 seconds, but every smartphone deadtime lasts at least 19200 seconds.

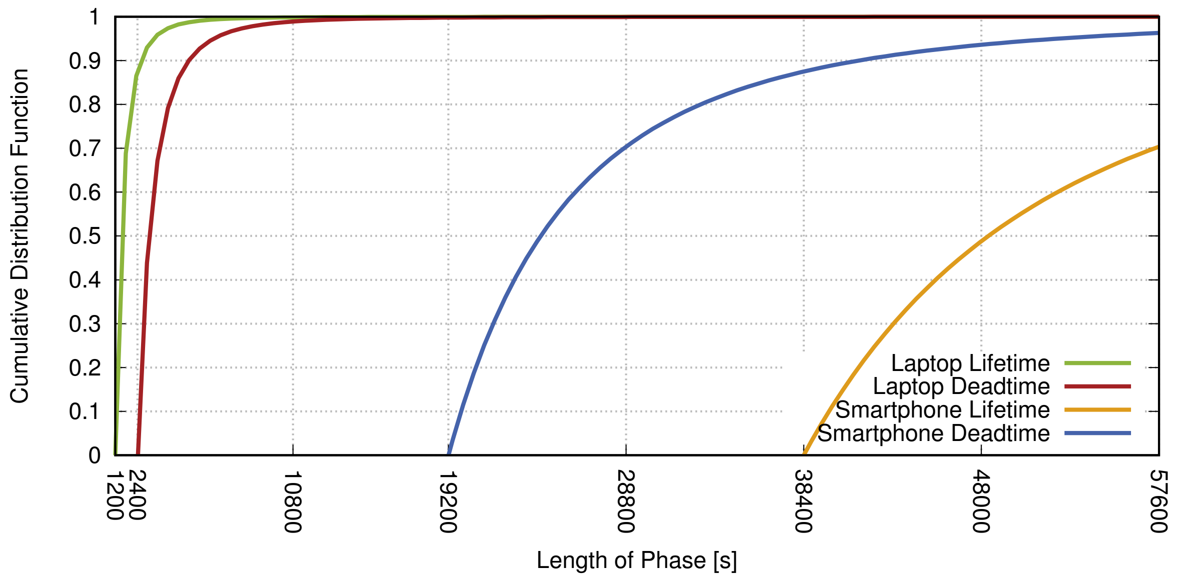


Figure 8.4: *Cumulative distribution functions for pareto-distributed lifetime and deadtime phase lengths with the expected values from Table 8.2 (shape parameter $\alpha = 3$)*

Capability model

High capability is reflected in SocioPath by a low **Communication Costs Value (CCV)** (see Chapter 5) and low capability is reflected by a high CCV. For the simulation scenarios, CCVs are fix for each device class. Each device class's CCV is shown in Table 8.2.

CCVs are abstract values which are chosen arbitarily here. For this evaluation, penalty values for being battery-driven (factor 5) and having a metered mobile connectivity where every sent and received byte counts against a quota (factor 5). However, this is configurable and the simulation environment also supports other values.

For the simulation scenarios, it is assumed that a smartphone is battery-driven and has a metered mobile connectivity, therefore the penalties sum up to a factor of 10. A laptop is battery-driven, but when it has connectivity, it is always a WiFi connection, therefore the overall penalty factor is 5. A home server is neither battery-driven and has fast and cheap connectivity, therefore the overall penalty factor is 1.

8.2.3 Device Ownerships

To the generated users, **application devices** and optionally **support devices** are mapped.

Application devices

An application device is a device which runs the *SodessonTestApp* with a yet to-be-defined application behaviour (see Section 8.2.4). This means, an application device publishes data objects and shall receive data objects if its user is a subscriber.

During each simulation run, each user has exactly one application device. An application device is either a smartphone or a laptop. All users have the same type of application device.

Support devices

A support device is a device that is associated to a user (in addition to the application device), but does not run the *SodessonTestApp*. It acts as an additional resource that can be used by SocioPath.

The following distributions of support devices with the following namings will be regarded throughout the evaluation.

- *NoSupp:* no user has any additional devices besides his application device
- *TOSuppServer*: topic owner (TO) has an additional home server
- *TOSuppSmartphone*: TO has one additional smartphone
- *TOSuppLaptop*: TO has an additional laptop

8.2.4 Applications

Two different use-cases will be regarded: **private instant messaging** and **group instant messaging**.

For instant messaging, one or multiple disjunct groups are defined per simulation run. In each group, one user publishes a small data object while the remaining users of that group are subscribers. This process is repeated multiple times, with different users being publishers, therefore simulating group-internal conversations.

To realize this scenario via UCNs, anchor user *A* first becomes the TO for a pre-defined number m of topics. Each of *A*'s contacts becomes an allowed publisher / allowed subscriber for exactly one topic and subscribes to that topic.

This results in m groups. Given that $n_{Users} - 1$ have been sampled before as the anchor user's (i.e. TO's) contacts and the TO is member of each group, this results in $\left(\frac{n_{Users}-1}{m} + 1\right)$ users per group.

For the simulations, these two extremes are regarded:

- **Private instant messaging:** the scenario here is a private conversation between TO *A* and one distinct contact. Thus, $n_{Users} - 1$ topics are generated by *A*. *A* subscribes to all of them and makes each contact an allowed publisher / allowed subscriber for a different topic. Each contact subscribes to his distinct topic. Every message with a given topic t is either published by A or the single distinct contact C_t associated to t. That message is only delivered to two subscribers: A and C_t, Thus, each group size is 2.

- **Group instant messaging:** the scenario here is a group conversation between TO A and all his contacts. Only one single topic is generated by TO A. A subscribes to it and defines all contacts as allowed publishers/allowed subscribers for this topic. Each contact subscribes to this topic. Any user can publish a message. The message gets delivered to all users. Thus, the size of the (only) group is n_{Users}.

Figure 8.5 shows both private and group instant messaging for user A and three contacts.

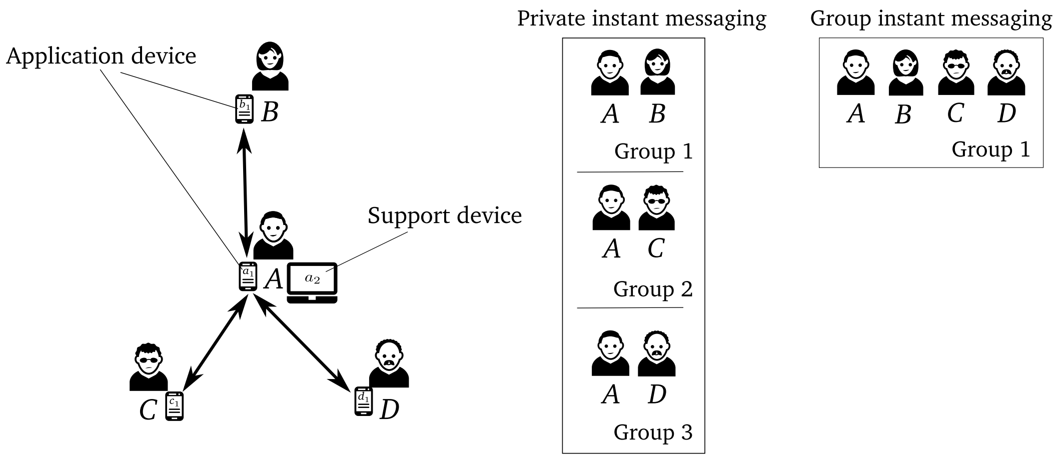


Figure 8.5: *Private and group instant messaging scenario for anchor user / topic owner A and three contacts.*

On each application device, an instant message is published on average every 200 seconds[2]. This average send period is derived by [3] where an average of 17.6 messages per hour was measured for participants in a study on instant messaging behaviour.

Every instant message published by contact X_t is a data object with the subscribed topic t. For each instant message published by A, one of the $(n_{Users} - 1)$ topics is picked randomly (uniformly distributed). Since A sends as many messages as a single contact, on average each contact receives $\frac{1}{n_{Users}-1}$ of the number of messages the same contact sends. In turn, A receives on average $n_{Users}-1$ times as many messages as A sends.

Each instant message size is uniformly distributed between 2 and 150 bytes. Thus, the expected value is 76 bytes. This number reflects the average of 13.5 words per instant message [58], the assumption of an average of 5 characters per word in the English language [98] and a UTF-8 encoding which mostly uses Western 1-byte characters.

All instant messages are piggybacked in SocioPath NOTIFY$_{\text{Req}}$ messages.

[2]This average is reached by uniformly distributed intervals between 0 and 400 seconds.

8.3 Performance Metrics

During each simulation scenario, two metrics are evaluated: **delays** and **sending costs**.

8.3.1 Delays

From the time when an application on a publishing device has published a new data object until it has been received on a specific subscriber device, four notable points in time can be identified (see Figure 8.6):

- Time ① – Published: an application on the publishing device publishes the data object
- Time ② – Sending first possible: the publishing device is alive at ① or becomes alive for the first time after ①. Thus, it is able to forward the data object to any other devices
- Time ③ – Receiving first possible: the subscriber device is alive on or after ② and is potentially able to receive the data object if delivered by another device
- Time ④ – Received: the alive subscriber device has fully received the data object, i.e. got it completely delivered from at least one other device

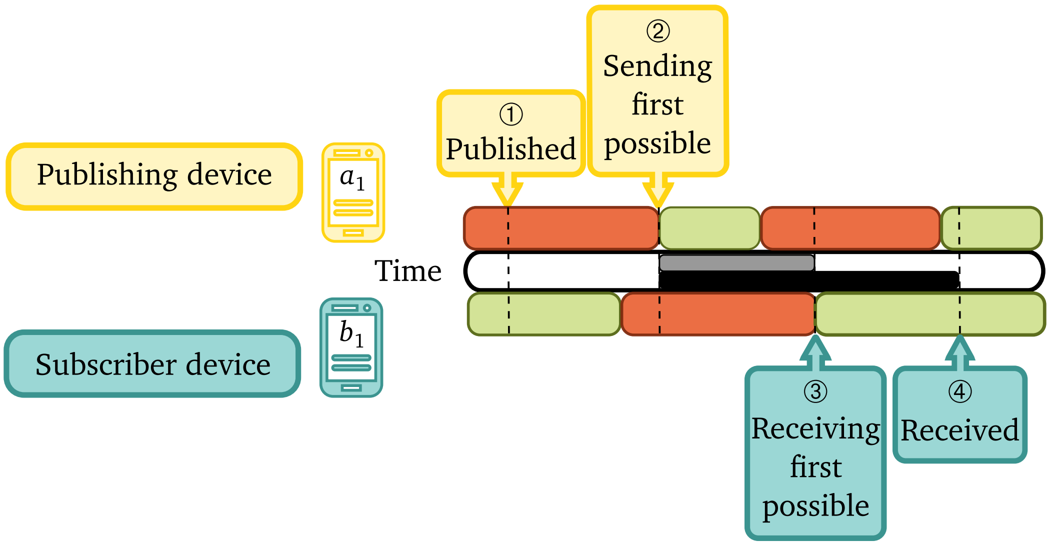


Figure 8.6: *A publishing and a subscriber device with alternating lifetime (green) and deadtime (red) phases. The important durations here are the baseline delay (grey, between ② and ③) and the delivery delay (black, between ② and ④)*

For each subscriber device, different values for ③ and ④ can occur. It is ① ≤ ② ≤ ③ ≤ ④.
If the publishing device is alive at the time of publishing, it is ② = ①.
From these points in time, two notable delays relevant for the evaluation can be deduced:

- **Baseline delay**: time period between ② and ③
- **Delivery delay**: time period between ② and ④

Baseline delay

The baseline delay BD(a_1,b_1,δ_t) is a lower bound for the delay of delivering a data object δ_t from the publishing device a_1 to a specific subscriber device b_1.

The baseline delay covers the timespan where publishing device a_1 is first able to forward δ_t to any other device until the time where subscriber device b_1 is first able to receive data objects from any other device, i.e. from ② to ③. Thus, the baseline delay is only defined by the lifetime and deadtime phases of the publishing and subscriber device respectively.

This is the smallest delay considered achievable. To see this, imagine a CSP which is always available and any other device can forward or receive a data object of arbitrary size in no time as long as that device is alive. Therefore, the publishing device a_1 would forward the data object as soon as possible (at ②) and the subscriber device b_1 would receive it as soon as possible afterwards (at ③).

Physical transmission delays are ignored here. The baseline delay is therefore a theoretical lower bound which cannot be achieved in reality: If a_1 and b_1 are both alive during the time where sending is possible for a_1, it is ② = ③. Hence, BD(a_1,b_1,δ_t) is exactly 0 seconds in this case.

Delivery delay

The delivery delay DD(a_1,b_1,δ_t) covers the timespan where the publishing device a_1 is first able to forward δ_t to any other device until the time where a specific subscriber device b_1 has fully received δ_t, i.e. from ② to ④.

It is ④ ≥ ③, thus it is DD(a_1,b_1,δ) ≥ BD(a_1,b_1,δ).

Unlike the baseline delay, the delivery delay is not a theoretical lower bound. Instead, the delivery delay adds up any delays that occur after ② until ④ is finally reached. The expected main contributors are as follows:

- Network latency between any two devices
- Selection of involved devices as given by the Decision Engine
- Deadtimes of all devices involved in the delivery, i.e. publishing device, subscriber device, TO devices or any other support devices.

Example

The following example shall illustrate how baseline delay and delivery delay depend on the publishing device's and each subscriber device's individual lifetime and deadtime phases.

Figure 8.6 shows two devices: one publishing device a_1 and a subscriber device b_1 which is supposed to receive a data object δ_t published by a_1. At least one of the two devices is a TO device.

At time ①, an application on a_1 publishes δ_t. At the same time, a_1 is dead, therefore δ_t cannot be delivered to b_1 (nor any other device, if there would be any). Still, from the application's point of view, the message has been successfully published and passed to the SODESSON middleware.

At time ②, a_1 becomes alive for the first time after ①. From this point, it becomes possible for a_1 to forward δ to any other devices. Thus, it is always ① ≤ ②.

b_1 becomes alive at ③. It was dead at ②, thus it is now first possible for b_1 to receive δ_t. Note that b_1 was alive at ①. However, it is impossible for b_1 to be notified about nor receive δ until the publishing device a_1 is alive for the the first time. Thus, it is always ① ≤ ② ≤ ③.

At ⑤, b_1 has fully received δ_t. Most of the difference between ③ and ④ is due to a_1 being in a deadtime phase again. a_1 is the only source device in this scenario, therefore δ_t cannot be delivered to b_1 if a_1 is dead. After a_1 becomes alive again, an additional delay is assumed due to underlay network latency for the transport of the notification.

In Figure 8.6, the baseline delay BD(a_1,b_1,δ_t) is displayed as the grey bar in the middle. The delivery delay DD(a_1,b_1,δ_t) is displayed as the black bar in the middle.

8.3.2 Sending Costs

As a second performance metric, the costs for sending a data object will be measured. To this end, each sent byte of each device is measured and multiplied with the respective device's CCV as penalty. Thus, the costs of each device will be measured in **weighed bytes per second** or **wb/s**.

Based on these measurements, three values will be regarded in the statistics:

- Minimum: costs of the single device with the least costs in one simulation run
- Mean: average costs of all devices in one simulation run
- Maximum: costs of the single device with the most costs in one simulation run

Since multiple simulation runs are performed, the cost plots will show an *average minimum*, *average mean* and *average maximum*.

8.4 Comparison of Results

Based on the given evaluation metrics (Section 8.3), results can be compared in terms of *better* and *worse*.

8.4.1 Delays

In the upcoming simulations, baseline delay and delivery delay are measured for each published data object, between the publishing device and the application devices of each subscriber. Since each user has only one application device where he consumes the data object, this is the delay each user cares about. Support devices do not count towards baseline delay nor delivery delay.

The simulation results are aggregated in a cumulative distribution function (CDF) for baseline delays and delivery delays respectively. As already defined in Section 8.2.2, a CDF is defined as $F_X(x) := P(X \leq x)$. It describes the probability P that a random variable X will be less than or equal to x. Since a single delay can be regarded as a sample for a random variable X, the CDFs describe the percentage of delivery delays and baseline delays which are smaller or equal to x.

Via the CDF measurements for a given delay X, baseline delay and delivery delay can be compared to each other. This way, SocioPath's actual performance in terms of delivery delay can be assessed with regard to the theoretical lower bound which is the baseline delay: the smaller the difference between the two, the *better*.

Only delays lower than 8 hourse (28800 seconds) will be taken into account, which is the expected deadtime of a smartphone. Since not the absolute values of the delivery delays are relevant, but instead their difference to the baseline delay, the results thus display how the delivery delays compare to the baseline for the first 8 hours. This is deemed to be sufficient for a good performance evaluation: if a large gap between baseline and delivery line cannot be closed within 8 hours, performance will be unacceptable anyway.

8.4.2 Sending Costs

Generally, the lower the measured average minimum / average mean / average maximum costs, the *better*. Additionally, the device with the highest costs will be identified. This way it is possible to tell which user will have the highest costs with what device.

8.4.3 Privacy

According to the privacy model in Section 3.6, a higher privacy level is *better* since it gives away less application metadata to the users of a closed group. Instant-to-All and Offload-First reach privacy level IV, i.e. the highest achievable privacy level according to the model.

8.5 Simulation Runs

Each simulation run is measured for 48 hours. This decision is based on the long expected lifetimes (16 hours) and deadtimes of smartphones (8 hours). Shorter simulations run would lead to the risk that one smartphone is always alive or always dead during a whole run. Before the 48 hours are measured, there is a 24 hour transition phase where all subscriptions, contacts, multiple devices per user are established and made known to the relevant devices. This "warm-up" does not count to the statistics. This way, an stabilized communication scenario is simulated. This excludes for example outliers in the delivery delays. Assume *SodessonTestApp* subscribes to a topic, the device is stored as a subscriber device in the *GlobalSodessonTestObserver* and counted for delivery delays, but the subscription request takes a long time until it reaches a TO device and before data objects get delivered. Such transition cases are not regarded here.

For each configuration, 100 simulations runs are performed. Of the results, the average is calculated, as well as the 98th percentile. The latter is shown as confidence intervals in all plots below. The respective lines and bars show the average.

8.6 Private Instant Messaging

First, private instant messaging will be regarded (see Section 8.2.4). Each closed group contains the TO and one distinct contact. The TO and the contact each have exactly one application device. All application devices in the same simulation run have the same class – they are either smartphones or laptops.

Additionally, there is either no support device (*NoSupp*) or exactly one TO support device (*TOSuppServer, TOSuppSmartphone, TOSuppLaptop*).

First, the delays for Instant-to-All (Section 8.6.1) and Offload-First (Section 8.6.2) will be evaluated. State repairs are reactive here. For comparison, delays with periodic state repairs will be shortly discussed in Section 8.6.4.

Second, the costs for Instant-to-All (Section 8.6.5) and Offload-First (Section 8.6.6) will be evaluated.

An summarizing overview on the key results will close this section.

8.6.1 Delays: Instant-to-All

Here, the baseline and delivery delays for private instant messaging with Instant-to-All will be evaluated.

Baseline delays for laptops and smartphones

Figure 8.7a shows graphs for laptops as application devices. For the baseline delay CDF (red line), it is $F_X(0) = 0.35$, i.e. about 1/3 of all published instant messages have a baseline delay of zero. This matches exactly with the expectation that a laptop is alive 1/3 of the time: at the time when sending is first possible for the publishing device, the subscriber device is alive at the same time with a probability of 1/3, resulting in a baseline delay of zero.

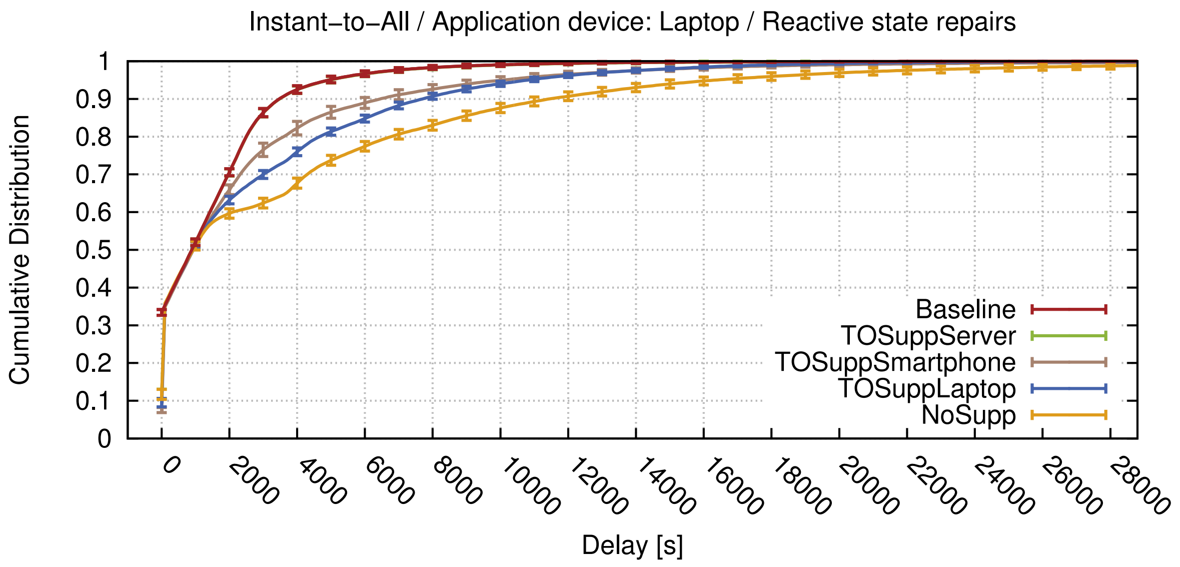
A similar observation can be made for smartphones (Figure 8.7b, red line). A smartphone is expected to be alive 2/3 of the time. For the baseline delay CDF, it is $F_X(0) = 0.72$, which is only slightly above the expected value of 2/3.

For laptops, the baseline delay CDF eventually reaches 1.0 in Figure 8.7a at about 20000 seconds. For smartphones on the other hand (Figure 8.7b), only $\sim$ 95% of all baseline delays are lower than eight hours. This is an important result: even though smartphones are alive more often than laptops (2/3 of the time compared to 1/3), smartphones are expected to have much longer deadtime phases – according to Figure 8.4, $\sim$ 30% of all smartphone deadtimes are longer than 28800s. Thus, once a smartphone is dead, it is less probable to become alive again than a laptop. While it is dead, it cannot receive any new instant messages, which results in longer baseline delays.

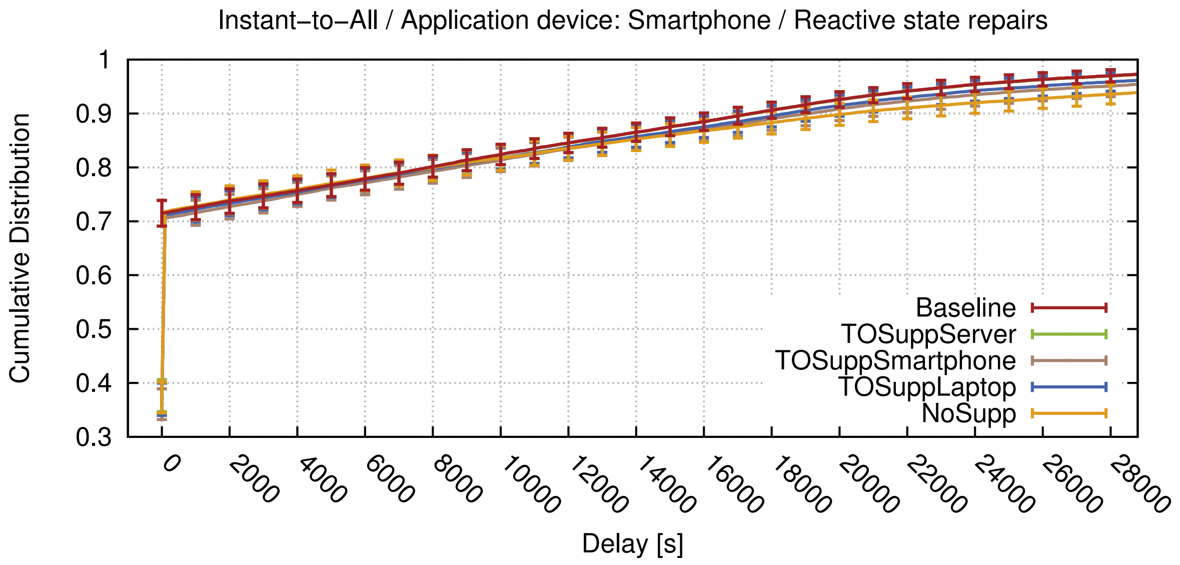
Delivery delays with no support device

For laptops as application devices and *NoSupp* (yellow curve in Figure 8.7a), low delivery delays until $\sim$ 1800 seconds are about as probable as for the baseline, i.e. the best case. For higher delivery delays, a gap between delivery delay CDF and baseline delay CDF opens and gradually closes again.

This can be explained as follows: in private instant messaging, there is only one publishing device and one other subscriber device per data object. The baseline regards the interval from the time when the publishing device is first alive to the time when the subscriber device is first



(a)



(b)

Figure 8.7: *Private instant messaging / Instant-to-All: delay CDFs with reactive state repairs*

alive. After 1800 seconds, more and more cases occur where the subscriber device becomes alive, but the publishing device is already dead again and cannot deliver the data object – note that the expected lifetime phase for a laptop is 1800 seconds. Afterwards, it can happen that the publishing device becomes alive again, but the subscriber device is dead again and so on. As soon as one device is alive and the second one just becomes alive, the second device initializes a state repair reactively and the data object is successfully delivered eventually. The probability that both devices are at least once alive at the same time increases as time passes, therefore the gap between baseline delay CDF and delivery delay CDF closes again.

For smartphones as application devices and *NoSupp* (yellow curve in Figure 8.7b), the delivery delay CDF behaves in exact the opposite way as laptops: instead of a gap which opens and then closes again, here the difference between delivery and baseline delay CDF grows for higher delays. This can be explained as follows: smartphones are more often alive than laptops (2/3 of the time as opposed to 1/3). Thus, there is a higher probability that the publishing device is still alive when the subscriber device becomes alive for the first time. This explains the general closeness between delivery delay CDF and baseline delay CDF. However, once a smartphone is dead, it can be dead for a long time – according to Figure 8.4, 30% of all smartphone deadtimes are longer than 28800 seconds. Therefore, most state repairs happen more sooner than later. Once the publishing device becomes dead, the probability that it becomes alive again before the end of the observed 28800 seconds is much lower than for laptops. Thus, the probability for successful state repairs diminishes here, the more time passes and no state repair has been successfully performed yet.

Around 0 seconds, Figures 8.7a and 8.7b seemingly show a jump of the delivery delay CDF to the baseline delay CDF. Figure C.1 in Appendix C shows the first 5 seconds for delivery delay CDF and baseline delay CDF in larger detail. It explains the jump: the baseline is a theoretical lower bound that can equal zero, but delivery delays are always larger than zero due to the simulated network latencies in OverSim. After 1-2 seconds, the delivery delay CDF has approximated the baseline delay CDF.

Delivery delays with TO support device

For laptops, TO support devices help to close the gap after 1800 seconds faster (see Figure 8.7a). Instead of making a state repair directly between publishing and subscriber device, the publishing device can make a state repair with the support device. Later, the subscriber device can make a state repair with the support device. In this case, publishing and subscriber device do not have to be alive at the same time. Naturally, the more often a support device is available for state repairs, the more helpful it is. This is reflected by the increasing closeness to the baseline from *NoSupp*, over *TOSuppLaptop*, *TOSuppSmartphone* up to *TOSuppServer*. The is latter always available, and thus almost congruent with the baseline and hardly visible in the graph.

To see this, also refer to Table 8.3. It shows selected values from the CDF plot, i.e. for delays lower or equal to 5s, 3600s and 28800s respectively. For example: for laptops with *NoSupp*, 64,6% of all delivery delays are lower than or equal to 3600 seconds. With *TOSuppServer*, it is 90,6%. The baseline is 90,7% here.

For smartphones, support devices do not help as much as for laptops. This can also be explained by the long lifetimes and long deadtimes of smartphones – support devices eliminate the requirement that publishing and subscriber device are alive at the same time. If both smartphones are alive at the same time, a support device is unneeded. If the subscriber smartphone is dead and never comes alive again, a support device cannot help either – this is the reason why the gap widens . Thus, smartphones do not profit from support devices as much as laptops do.

Again, this can be seen in Table 8.3. For example: for smartphones with *NoSupp*, 75,5% of all delivery delays are lower than or equal to 3600 seconds. The baseline is 75,6% here, i.e. the delivery delays with *NoSupp* are already very close, even without a support device.

Table 8.3: *Private instant messaging / Instant-to-All: percentage of delivery delays that are lower or equal to 5s / 3600s / 28800s*

	Private instant messaging / Instant-to-All					
	Application Device: Laptop			**Application Device: Smartphone**		
	$\leq$ 5s	**$\leq$ 3600s**	**$\leq$ 28800s**	**$\leq$ 5s**	**$\leq$ 3600s**	**$\leq$ 28800s**
Baseline	**35.0%**	**90.7%**	**99.99%**	**71.7%**	**75.7%**	**97.2%**
TOSuppServer	34.3%	90.6%	99.93%	71.4%	75.2%	97.2%
TOSuppSmartphone	33.9%	79.9%	99.66%	70.5%	74.5%	95.4%
TOSuppLaptop	34.5%	73.1%	99.87%	71.0%	74.9%	96.1%
NoSupp	34.0%	64.6%	98.84%	71.6%	75.6%	93.9%

8.6.2 Delays: Offload-First

Now, the most relevant differences of Offload-First as a Decision Engine compared to Instant-to-All will be discussed. Besides the Decision Engine, the same configuration will be discussed.

Baseline delays

In general, the baseline delays do not differ from the ones where Instant-to-All is used. Baseline delays are only defined by the lifetimes and deadtimes of each publishing device and respective subscriber device. They are independent from the Decision Engine and therefore do not have to be discussed again. To see this, compare the red curves of the respective plots between Figure 8.7 and Figure 8.8.

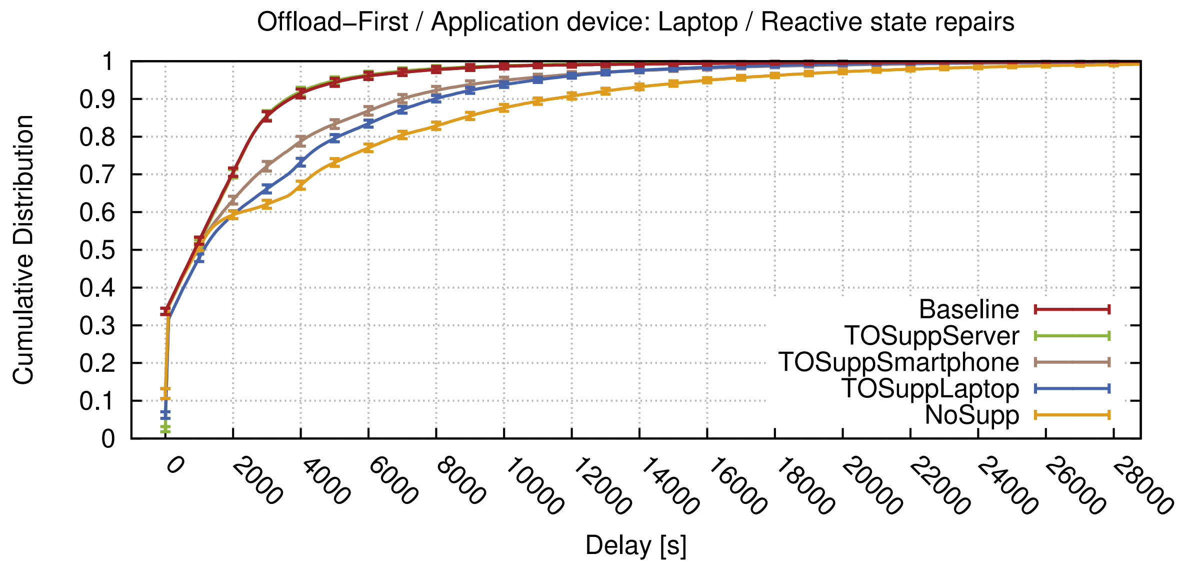
Delivery delays with no support device

If there are no support devices, i.e. only there is exactly one subscriber device besides the publishing device for each data object, the delivery delays for Offload-First are the same as for Instant-to-All: the publishing device sends a NOTIFY message to the subscriber device and asks it to forward to all other devices of the same user. Since there are none, the notification process is complete, just like for Instant-to-All. State repair procedures are also identical if there are no support devices. To see this, compare the yellow curves of the plots in Figure 8.7 and Figure 8.8 respectively.

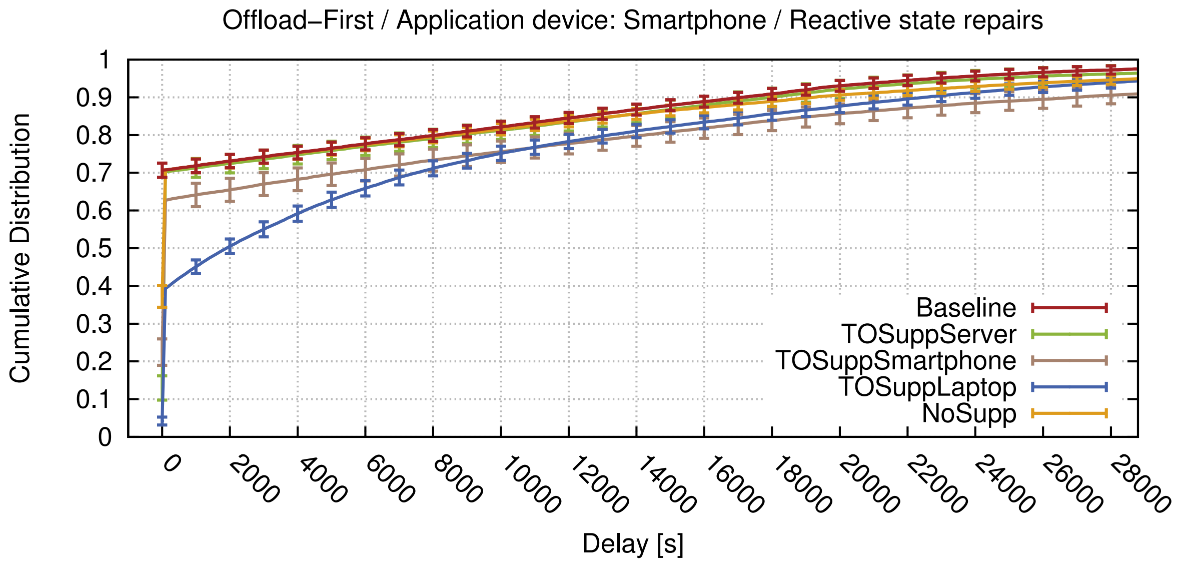
Delivery delays with TO support device

If the TO support device has lower or equal communication costs to the application devices, it becomes a bottleneck for forwarding notifications in Offload-First, unlike in Instant-to-All. If this support device is dead, the whole notification process stalls and state repairs are required later, even if the application devices are alive.

This bottleneck effect can be observed in Figure 8.8b (Offload-First), as opposed to Figure 8.7b (Instant-to-All). Here, each user has a smartphone as the application device. For the blue curve, there is a *TOSuppLaptop*. Since the Communication Costs Value (CCV) is lower for laptops (5)



(a)



(b)

Figure 8.8: *Private instant messaging / Offload-First: delay CDFs with reactive state repairs*

than for smartphones (10), Offload-First chooses the *TOSuppLaptop* as the device for offloading. Only the *TOSuppLaptop* received the NOTIFY message first. However, laptops are less often alive than smartphones. If the *TOSuppLaptop* is dead when the publishing device tries to send the NOTIFY message, the whole notification process stalls until the next state repair betwen these two devices. Unlike for Instant-to-All, the other application device does not get notified directly.

Due to this bottleneck effect when using support devices, the delivery delay CDF is visibly worse than for Instant-to-All. The only exception is *TOSuppServer* which has no deadtimes at all and is thus always available.

8.6.3 Delays: Scalability

In the previous results, all simulations were done with 20 users. Since private instant messaging was evaluated, this means there were 19 groups with the topic and one distinct contact. Figure 8.9 shows results for other group sizes: 5, 50 and 100 users for of Instant-to-All with *TOSuppServer* and *NoSupp*.

The results indicate that the delays are independent from the number of users: there are two clusters of graphs, but the difference is made by *TOSuppServer* versus *NoSupp*. The graphs align with the respective graphs in Figure 8.7a, which was already discussed in Section 8.6.1.

User numbers not having an effect here is an intuitive result: independent from the number of users, each group size is still 2. This means, each published instant message only needs to be delivered to the TO's application device and one distinct contact's application device. This is with a single state repair between the devices of the respective users. The only difference is that the number of state repairs for the topic owner changes with the number of contacts.

This shows that in terms of delay, the private instant messaging scenario scales at least to up to 99 contacts of the topic owner.

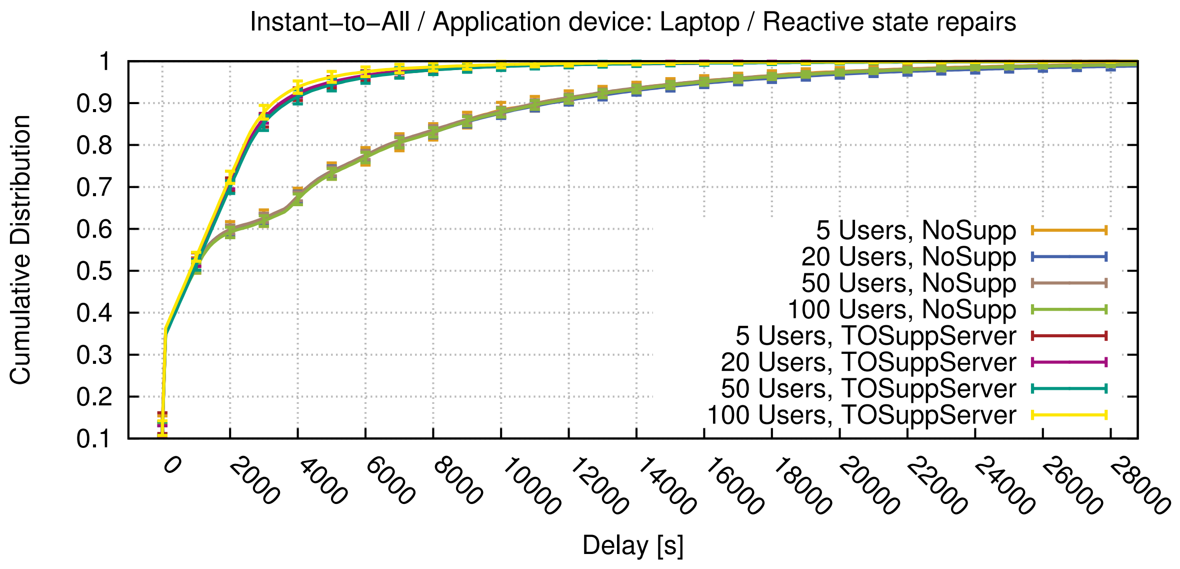
8.6.4 Delays: Periodic State Repairs

In the previous results for delivery delays, state repairs were *reactive*: as soon as a device becomes alive after a deadtime, it initiates state repairs and also demands a returned $\text{STATE}_{\text{Req}}$ from the contacted devices. An alternative to reactive state repairs are *periodic state repairs*: each device can simply periodically initiate state repairs with all known devices, regardless of any previous deadtimes. Here, no returned $\text{STATE}_{\text{Req}}$ messages are demanded, since every device sends them periodically. The periodic interval is a system-wide parameter and is called State Sending Interval (SSI).

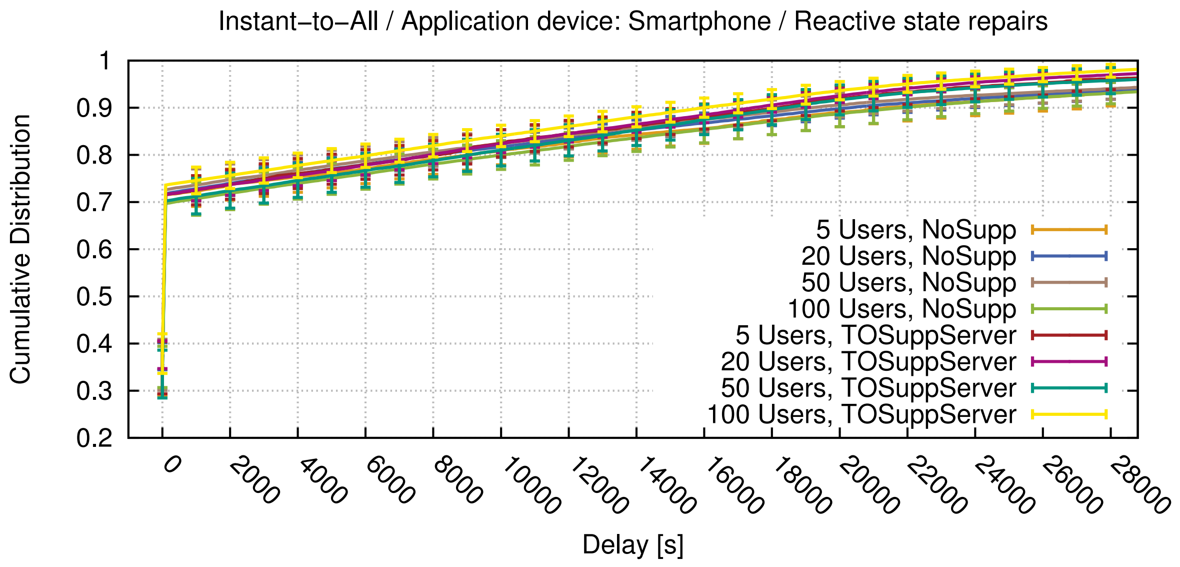
Generally, the sooner a state repair is performed, the better: the sooner states are repaired, the sooner missed notifications are delivered to the subscriber device. Less frequent state repairs have a negative impact on the delivery delay CDF. It now depends on the class of the application devices, whether reactive or periodic states should be preferred. This shall be discussed at the example of Offload-First for both laptops and smartphones.

Laptops have short lifetimes and short deadtimes. Thus, a laptop often changes from a deadtime to a lifetime. If reactive states are used, state repairs are also often initiated here. Figure 8.10a shows periodic SSIs for laptops. $\text{STATE}_{\text{Req}}$ messages are sent on average every 500s. Compare Figure 8.8a (reactive states) to Figure 8.10a (periodic states). If periodic states are used, there is a larger gap between the delivery delay CDFs and the baseline CDF. For laptops, reactive state repairs are on average faster than periodic state repairs every 500s.

However, smartphones have long lifetimes and long deadtimes. Thus, a smartphone rarely changes from a deadtime to a lifetime. If reactive states are used, state repairs are very rare. Thus, smartphones profit from periodic state repairs. To see this, compare Figure 8.8b (reactive states) with Figure 8.10b (periodic states). If periodic states are used, the delivery delay CDFs approximate the baseline delay CDF very closely and visibly faster than with the reactive approach. Especially the availability bottleneck *TOSuppLaptop* is alleviated in Offload-First, due to the periodic state



(a)



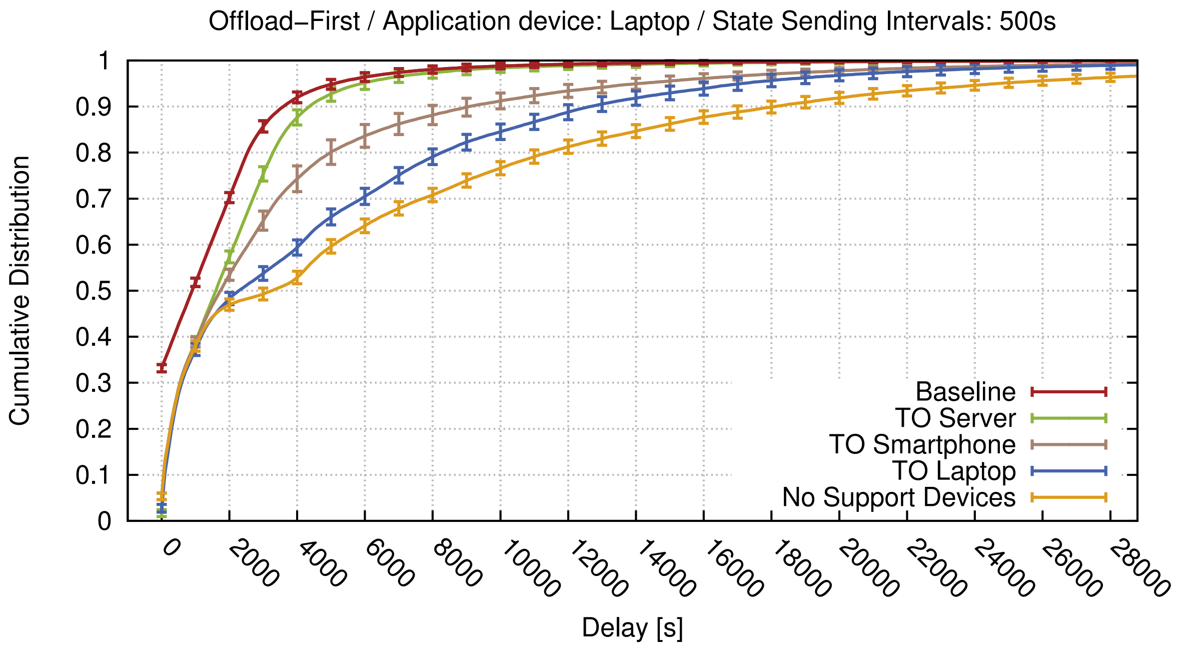
(b)

Figure 8.9: *Private instant messaging / Instant-to-All: delay CDFs with different user numbers*

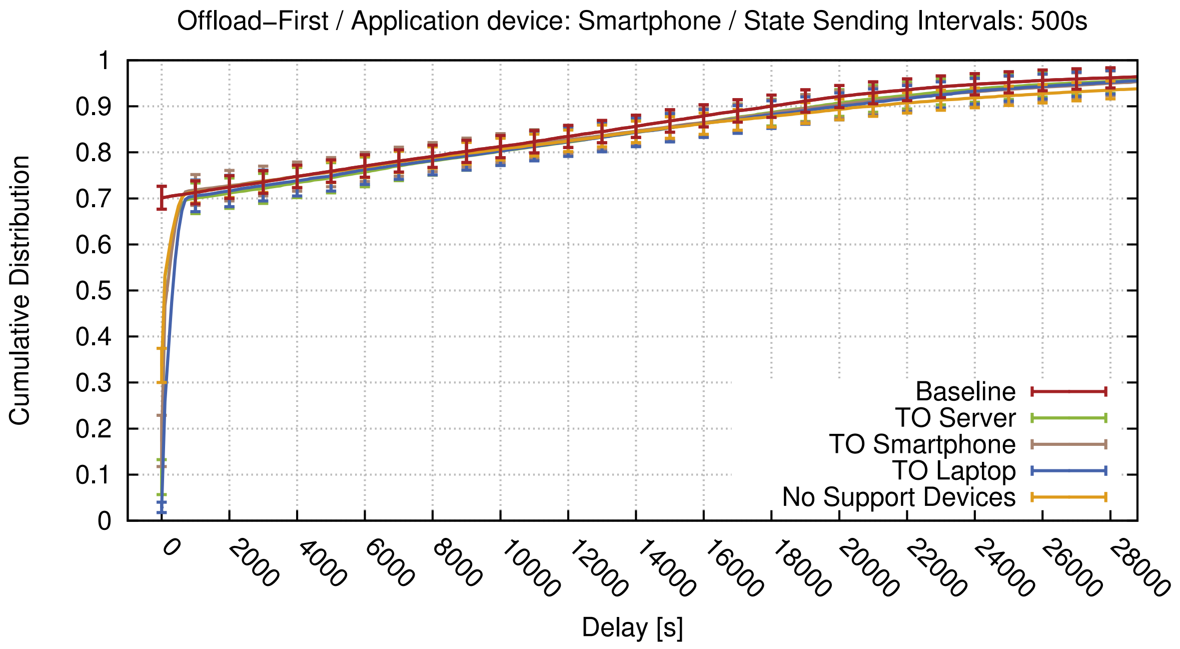
repairs of the application devices with each other: for SSI = 500s reactive, 70% of all data objects are delivered after 1000 seconds, as opposed to 8000 seconds with the reactive approach.

8.6.5 Costs: Instant-to-All

For Instant-to-All, the red-toned bars in Figure 8.11 show the total sent bytes per second, weighed with each device class's penalty value (weighed bytes per seconds or *wb/s*). For each device distribution, the minimum costs (taken from the device with the overall least costs), mean and



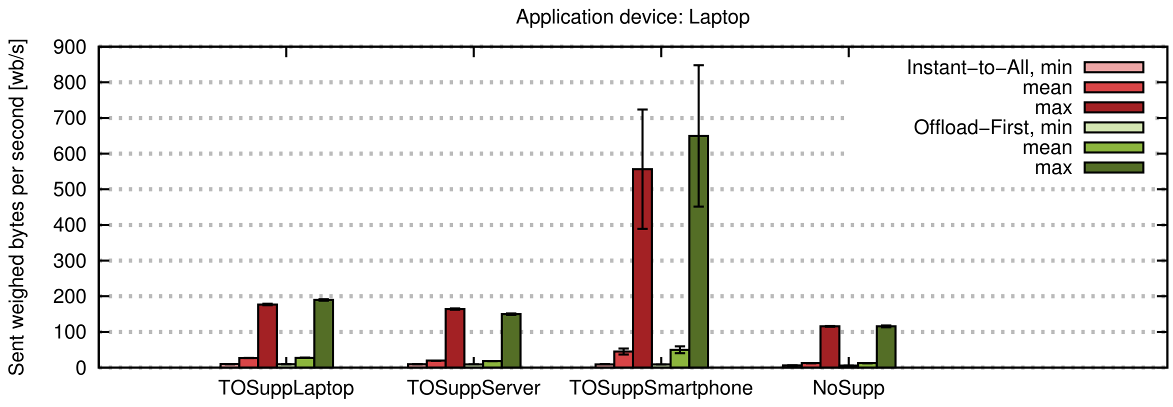
(a)



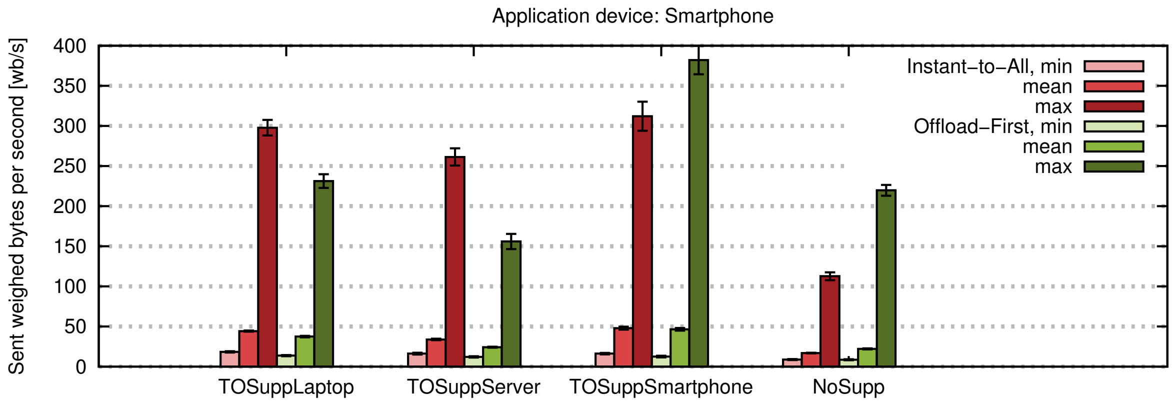
(b)

Figure 8.10: *Private instant messaging / Offload-First: delay CDFs with periodic state repairs*

maximum (taken from the device with the overall highest costs) are shown. All state repairs are reactive.



(a)



(b)

Figure 8.11: *Private Instant Messaging: Costs*

Costs with no support device

If no support device is used, the minimum, mean and maximum costs for laptops and smartphones are similar: for laptops, the min / mean / maximum costs are 6 / 12.6 / 115.6 wb/s. For smartphones, the min / mean / maximum costs are 8.7 / 16.9 / 112.6 wb/s). This is counter-intuitive, since the CCV for smartphones is twice as high as for laptops, but the ratios of the number are 1.45 / 1.34 / 0.97 instead of 2.0 respectively. On the other hand, the availability ratio of smartphones is twice as high as the laptops' availability ratio. Thus, less state repairs are required for smartphones. These two factors balance each other out to a certain degree.

Costs with TO support device, laptops as application devices

For laptops as application devices and any support device (*TOSuppLaptop*, *TOSuppServer* and *TOSuppSmartphone*), the minimum, mean and maximum costs are higher than with no support device (see Figure 8.11a).

It can be assumed that the device with the minimum costs belongs to a contact, while the device with maximum costs belongs to the TO, as will be discussed further below. Thus, the higher costs affect the contacts as well as the TO, which can be deduced from higher minimum costs and higher maximum costs: the average minimum costs are: 9.6 / 9.4 / 9.6 wb/s for *TOSuppLaptop / TOSuppServer / TOSuppSmartphone* compared to 6 wb/s without any support device. In Instant-to-All, each contact's laptop sends a notification after publishing to two TO devices: the TO's laptop (application device) and the TO support device. The costs are not twice as high, since state repairs of the TO devices can also be made between the TO devices themselves – in that case the contact device would only need to send one notification. The average maximum costs are 174.2 wb/s (164.2 wb/s) for *TOSuppLaptop* (*TOSuppServer*) compared to 115.4 wb/s without any support device. Thus, the additional device also results in higher costs for the TO.

Note that the confidence interval is very large for *TOSuppSmartphone* in Figure 8.11a – the 98th percentile ranges between 389 wb/s and 723 wb/s. Whether the maximum costs are in the upper or lower end of this range, depends on the individual lifetimes and deadtimes of the single *TOSuppSmartphone*: once the smartphone is dead, it cannot be a target for state repairs for probably a long time. In this case, state repairs happen only between the laptops as application devices with their lower CCV. Since *TOSuppSmartphone* is not involved here, this reduces its costs, i.e. the maximum costs, significantly. However, if *TOSuppSmartphone* has a very long lifetime during the simulation, it often becomes a target for state repairs. This results in high costs for *TOSuppSmartphone*, i.e. high maximum costs.

Costs with TO support device, smartphones as application devices

For smartphones as application devices and *TOSuppSmartphone* (see Figure 8.11b), the costs also grow with a TO support device, as already discussed for laptops.

The confidence intervals for *TOSuppSmartphone* have a smaller range: regardless of the individual lifetimes and deadtimes of the support device *TOSuppSmartphone*, state repairs always happen between smartphones with a high CCV. This results in a lower variance for the maximum costs. Additionally, the average maximum costs are lower for smartphones as application devices than for laptops (compare *TOSuppSmartphone* in Figure 8.11b with *TOSuppSmartphone* in Figure 8.11a): the average maximum costs for smartphones (laptops) are 317 wb/s (615 wb/s). The reason for this result is again the higher availability of smartphones: less state repairs are required, which results in lower maximum costs.

Device with highest costs, with no support device

In all cases, there is a strong difference between mean and maximum, regardless of support devices and Decision Engine. This lets one suspect that a single device has most of the costs: the mean is measured across all devices, but the maximum is measured from a single device. In case of a completely fair distribution across all devices, the mean costs would equal the maximum costs. On the other hand, the stronger the difference between mean and maximum, the more one single device obviously behaves differently than the other devices regarded as a whole. This effect shall be investigated now.

To identify the single device with the highest costs for laptops as application devices, see Figure 8.11a with *NoSupp*. Here, the mean costs for each device are 12 wb/s, which for 20 devices yields an overall sum of 240 wb/s. Given that one device has costs of 115 wb/s (i.e. the maximum), this means that one single device has almost as high costs as all other devices taken together.

Intuitively, this has to be the application device of the TO. However, that device's publishing behaviour is identical to all contact's devices, i.e. on average it publishes as many instant messages as any other application device. For each instant message, only one one subscriber device must be notified – this is also identical to the costs of any contact's application device. Thus, the reason for the large difference must be state repairs. Indeed, when the TO device becomes alive from a deadtime, it performs up to 19 state repairs – one with each contact device that is currently alive. Vice-versa, each contact's device performs only one state repair, again with the TO device. Thus, besides its own large number of state repairs, the TO device is also the target of any other state repairs.

Device with highest costs, with TO support device

It is now clear that the device with the highest costs is a TO device. If there is a TO support device, the TO has two devices: the application device and one support device. It shall now be investigated which of the two TO devices has the highest costs. The respective device class could be deduced by comparing the maximum costs with the maximum sent bytes (unweighed). The results of this comparison are listed in Table 8.4.

Table 8.4 shows that the device which sends the most bytes is not necessarily the device with the highest costs. For example, for laptops as application devices and a TOSuppServer as a support device, a server sends most bytes. This can only be the TOSuppServer. However, when weighed with the CCV (server = 1, laptop = 5), a laptop has the highest costs. This has to be the laptop of the TO. As discussed above, Figure 8.11a shows for Instant-to-All and laptops as application devices, that a *TOSuppServer* raises the maximum costs. Thus, a *TOSuppServer* does not lower the costs, but instead raises them for the TO's application device.

The same considerations can be applied to all other device distributions as well, with one exception: for laptops as application device and *TOSuppSmartphone*, the support device actually has the highest costs.

Table 8.4: *Private instant messaging / Instant-to-All: Device class of the device with the most sent bytes (bytes per second, unweighed) and the highest costs (weighed bytes per second)*

	Private instant messaging / Instant-to-All			
	Application Device: Laptop		**Application Device: Smartphone**	
Support Device	**Max Sent Bytes (b/s)**	**Max Costs (wb/s)**	**Max Sent Bytes (b/s)**	**Max Costs (wb/s)**
TOSuppLaptop	Laptop	Laptop	Laptop	Smartphone
TOSuppServer	Server	Laptop	Server	Smartphone
TOSuppSmartphone	Laptop	Smartphone	Smartphone	Smartphone
NoSupp	Laptop	Laptop	Smartphone	Smartphone

8.6.6 Costs: Offload-First

For Offload-First, the green-toned bars in Figure 8.11 show the total sent bytes per second, weighed with each device class's penalty value (weighed bytes per second or *wb/s*). Most of the observations for Instant-to-All also apply here. Thus, only the relevant differences shall be discussed here.

Costs with no support device

There is a notable difference for the maximum costs for laptops and smartphones, when compared to Instant-to-All. For laptops, the maximum costs are about equal to Instant-to-All (see Figure 8.11a). For smartphones however, the maximum costs are about twice as high as for Instant-to-All (see Figure 8.11b). This difference can be explained by the way how original notifications (i.e. no resent notification during state repairs) are sent in Offload-First. If a contact's application device publishes an instant message, the following steps occur: first, the contact application device sends a notification to the TO's application device. Second – since the contact's application device CCV is lower or equal (here: equal) to the TO device's CCV – the TO device *sends back* the notification and asks the contact's device to forward it to other devices of the same contact. Since the contact has no other devices, this is an unneeded step that results in wasted costs (see also Table 8.5). This is an issue in the current design which can be easily resolved since the TO device knows that the publisher only has this single device.

For laptops as application devices, this effect is not visible. The reason for this is the low availability of laptops. Most notifications are (re)sent during state repairs, where no forwarding takes place. Offload-First behaves identically to Instant-to-All for state repairs.

Costs with TO support device

With laptops as application devices, results are comparable to Instant-to-All. For smartphones however, minimum, mean and maximum costs can be lowered with Offload-First. The respective values from Figure 8.11b are also displayed in Table 8.5. For the minimum and mean, the results show that it is always more cost-efficient to use Offload-First for smartphones and any TO support device. Offload-First can save 25.9% for the minimum and 28.7% for the mean (both with *TOSuppServer*).

For the maximum however, Offload-First is only preferable if the TO support device has a lower CCV than the application devices. In that case, the maximum costs can be lowered by 22.4% (*TOSuppLaptop*) and 40.3% (*TOSuppServer*) respectively. With *TOSuppSmartphone*, the maximum costs are 22.4% higher when using Offload-First.

As can be seen from Table 8.6, the device which sends the most bytes is not necessarily the device with the highest costs. This discussion was already led for Instant-to-All in Section 8.6.5. For Offload-First this means, that e.g. for smartphones as applications and *TOSuppServer*, still the TO's smartphone has the maximum costs – even though most bytes are sent by *TOSuppServer*. However, Offload-First relieves the smartphone from 40.3% of its costs as compared to Instant-to-All.

Table 8.5: *Private instant messaging: Comparison of minimum, mean and maximum costs between Instant-to-All and Offload-First. Application devices are smartphones. Absolute numeric values are in wb/s. Lower values are better.*

	Application Device: Smartphone		
Support Device	**Min Costs (wb/s) Instant-to-All**	**Min Costs (wb/s) Offload-First**	**Costs saved by Offload-First**
TOSuppLaptop	18.3	*13.7*	25.1%
TOSuppServer	16.2	*12.0*	25.9%
TOSuppSmartphone	16.1	*12.4*	22.9%
NoSupp	8.7	*8.6*	1.1%
	Mean Costs (wb/s) Instant-to-All	**Mean Costs (wb/s) Offload-First**	**Costs saved by Offload-First**
TOSuppLaptop	44.2	*37.4*	15.3%
TOSuppServer	33.8	*24.1*	28.7%
TOSuppSmartphone	47.8	*46.3*	3.1%
NoSupp	*16.9*	22.1	−30.8%
	Max Costs (wb/s) Instant-to-All	**Max Costs (wb/s) Offload-First**	**Costs saved by Offload-First**
TOSuppLaptop	297.8	*231.2*	22.4%
TOSuppServer	261.3	*155.9*	40.3%
TOSuppSmartphone	*312.1*	382.0	−22.4%
NoSupp	*112.6*	219.7	−95.1%

Table 8.6: *Private instant messaging / Offload-First: Device class of the device with the most sent bytes (bytes per second, unweighed) and the highest costs (weighed bytes per second)*

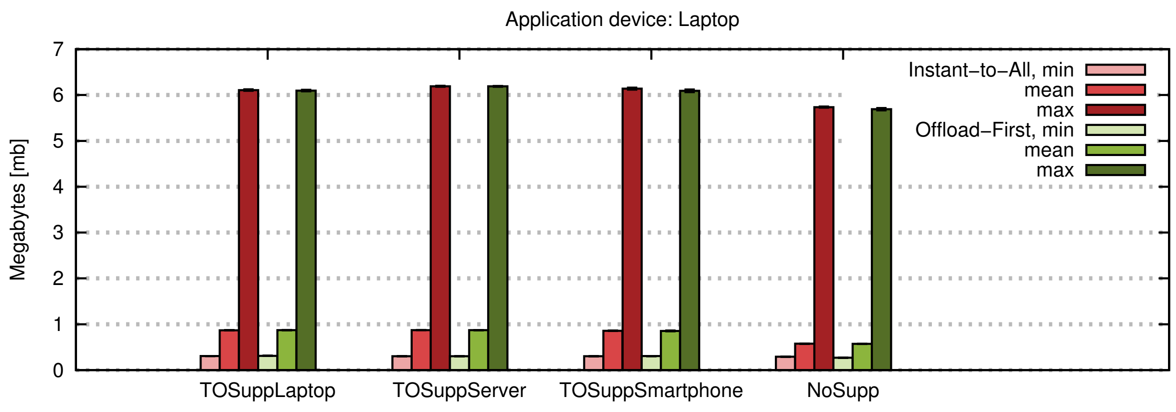
	Private instant messaging / Offload-First			
	Application Device: Laptop		**Application Device: Smartphone**	
TO Support Device	**Max Sent Bytes (b/s)**	**Max Costs (wb/s)**	**Max Sent Bytes (b/s)**	**Max Costs (wb/s)**
TOSuppLaptop	Laptop	Laptop	Laptop	Laptop
TOSuppServer	Server	Laptop	Server	Smartphone
TOSuppSmartphone	Laptop	Smartphone	Smartphone	Smartphone
NoSupp	Laptop	Laptop	Smartphone	Smartphone

8.6.7 Storage Footprint

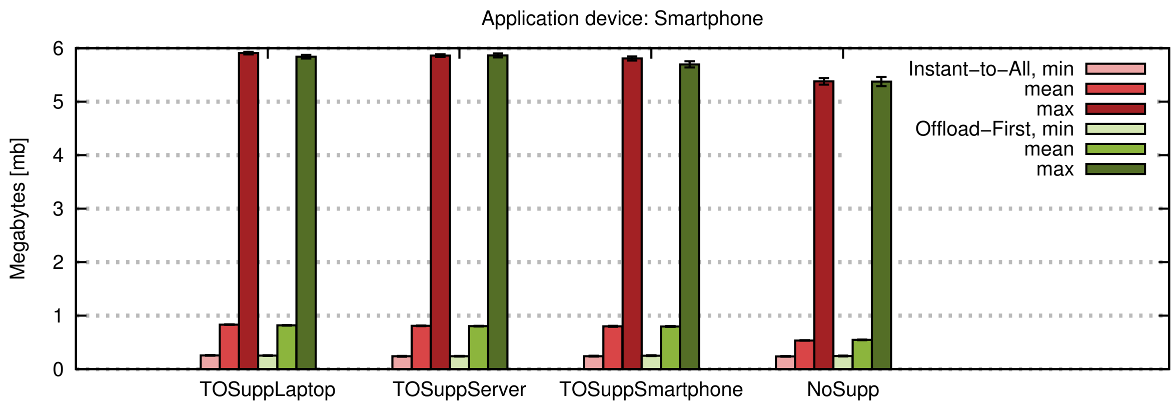
In order to estimate the storage footprint for the private instant messaging scenario, Figure 8.12 shows the minimum / mean / maximum size of the Local Data Storage (in unweighed megabytes) after the end of the simulation. Note that the Local Data Storage does not only contain the instant messages, but also all maintenance data objects, such as notifications about new subscribers, new devices, and so on.

Here it can be seen that for all device distributions, the final mean (maximum) size is below 1 (7) megabytes. Since 48 hours were simulated, the mean (maximum) would result in less than 15 (105) megabytes per month, which is deemed acceptable for the scenario. As already seen, the maximum is significantly higher than the mean, which indicates that again a TO device has the highest storage footprint.

The size of the Local Data Storage correlates with the delivery delay CDF, since each successfully delivered data object is an entry in the Local Data Storage. Here, it is only relevant whether a data object arrived within 28800s or not, thus only the far right of each delivery delay CDF graph needs to be taken into account. For example, since in Figure 8.7b (Instant-to-All, smartphones as application devices), the delivery delay CDF is worse for 28800s for NoSupp than for e.g. TOSuppServer. This is also reflected in the respective Local Data Storage size.



(a)



(b)

Figure 8.12: *Private instant messaging: Local Data Storage size at the end of simulation*

8.6.8 Key results

The following points sum up the most relevant insights from evaluating the private instant messaging scenario.

General observations

- Generally, it is possible to gain delivery delays that closely approximate the baseline.[3] This result applies to instant deliveries (i.e. delivery delays of a few milliseconds) as well as high delays, based on state repairs. This shows that both the designed notification and state repair systems work. In terms of delay, SocioPath is applicable for private instant messaging under conditions of the assumed model.

- If application devices have a low availability (here: laptops), TO support devices can improve the delivery delay CDFs for messages that could not be delivered instantly.[4] For example, with Instant-to-All and *TOSuppServer*, it is possible to deliver 90.6% of all messages in under 1 hour (baseline is 90.7%)[5], while only 64.6% are delivered with *NoSupp*.

- If application devices have a high availability (here: smartphones), TO support devices are not as useful, since the delivery delay CDF is very close to the baseline CDF, even without support devices[6].

- The sooner state repairs are performed, the better are the delivery delay CDFs.

- Not only the availability ratio, but also the expected lengths for lifetimes and deadtimes are relevant for both delivery delays and costs.

- If application devices have long lifetimes and long deadtimes, i.e. rarely become alive from a deadtime (here: smartphones), reactive states repairs are rarely performed. Here, periodic state repairs lead to better delivery delay CDFs than reactive state repairs.[7]

- Since the TO participates in 19 groups, but each contact only in one group, the costs for TO devices are much higher than for each contact's. For example, for Instant-to-All, *NoSupp* and laptops as application devices, the TO's laptop has as many costs as all contacts' devices taken together.

- Delivery delays CDFs are about identical for 5, 20, 50 or 100 users[8]. Thus, SocioPath scales for the given private instant messaging scenario up to 100 users in terms of delivery delays.

- Storage footprint is acceptable with less than 105 megabytes per month for the topic owner and less than 15 megabytes per month for contacts.[9]

[3]see Figures 8.7 and 8.8
[4]see Figures 8.7a and 8.8a
[5]see Table 8.3
[6]see Figures 8.7b and 8.8b
[7]compare Figure 8.7b with Figure 8.10b
[8]see Figure 8.9
[9]see Figure 8.12

Instant-to-All

- If Instant-to-All and no support devices are used, then high (low) availability and low (high) CCV cancel each other out with regard to costs: smartphones have a CCV which is twice as high as a laptop's CCV, but their availability ratio is also twice as high. Still, laptops and smartphones have comparable costs here. The reason is that high availiability requires less state repairs.
- For Instant-to-All, the advantage of an additional TO support device (improving the delivery delay CDFs) comes at additional costs for both the TO and the respective contact: each application device now has to send two notifications instead of one. The additional costs for the TO are defined by the TO support device's class. For example, a *TOSuppSmartphone* with its high availability is often a target for state repairs, which results in much higher sending costs for the TO due to its high CCV.

Offload-First

- In Offload-First, the TO device with the lowest CCV can become an availability bottleneck if it has a low availability: notifications get offloaded there first due to the low CCV. If it is dead, the whole notification process stalls.[10]
- For Offload-First, a support device can lower the costs. The average minimum / mean / maximum costs can be lowered up to 25.9% / 28.7% / 40.3% (smartphones as application devices, *TOSuppServer* as TO support device).[11] Since usually a TO device has the highest cost, the support device can help to relieve the cost of the application device. However, Offload-First brings the risk of availability bottleneck if the TO device with the lowest CCV has a low availability.

8.7 Group Instant Messaging

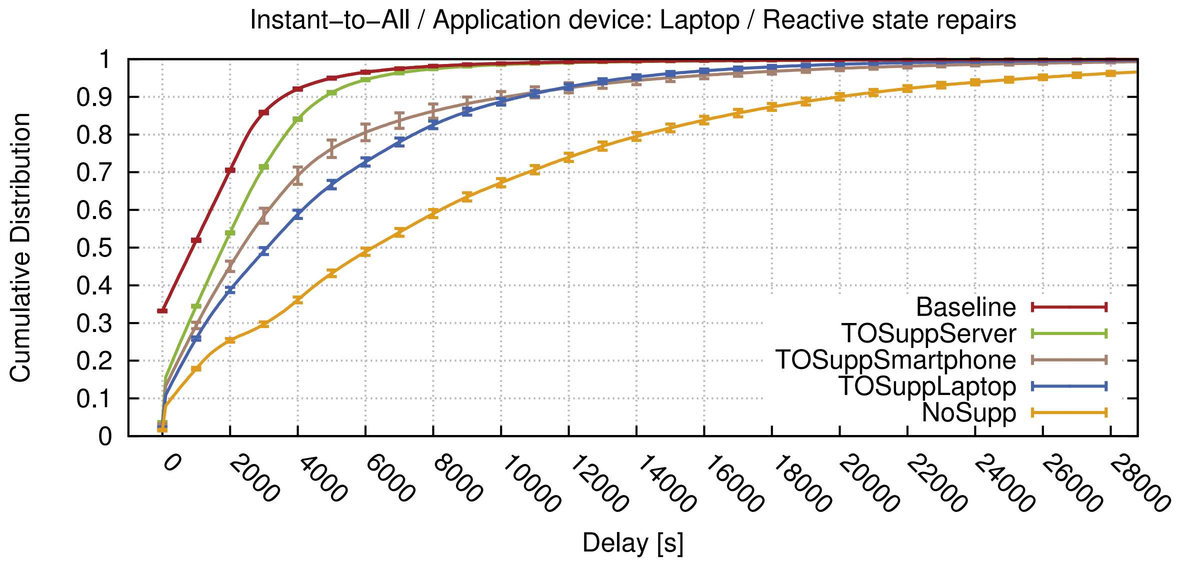
8.7.1 Delays: Instant-to-All

This section discusses the most notable differences of group instant messaging compared to private instant messaging with regard to delays when using Instant-to-All.

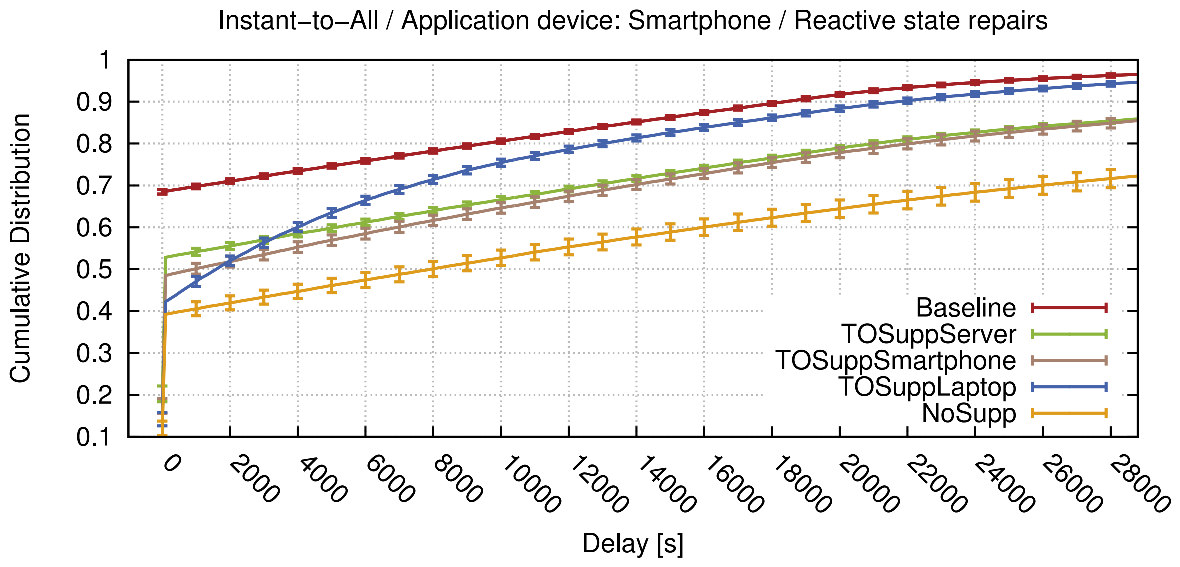
Baseline delays for laptops and smartphones

Baseline delays are identical to private instant messaging. To see this, compare the red curves between Figure 8.7a and Figure 8.13a (laptops) / Figure 8.7b and Figure 8.13b (smartphones). This result is to be excepted: the lifetime and deadtime distributions of the involved devices are identical in both private instant messaging and group instant messaging. The number of subscriber devices or Decision Engine does not have an influence here.

[10]see Figure 8.8b
[11]see Table 8.5



(a)



(b)

Figure 8.13: *Group Instant Messaging: Delay CDFs (Instant-to-All, reactive states)*

Delivery delays with no support device

For delivery delays on the other hand, the number of subscriber devices has a strong influence. For laptops and *NoSupp*, Figure 8.13a, shows that only 33% of all data objects are delivered in under 3600 seconds. This is a strong difference to private instant messaging where 64.6% are delivered in under 3600 seconds (see Figure 8.7a and Table 8.3).

For smartphones and *NoSupp*, Figure 8.13b, shows that 44% of all data objects are delivered in under 3600 seconds. This is higher than for laptops, which can be explained with the higher availability of smartphones. However, this is still a strong difference compared to the 75.6% for private instant messaging (see Figure 8.7b and Table 8.3).

The reason for this difference: TO devices are a more important bottleneck in the group instant messaging scenario than in private instant messaging. If one published data object cannot be delivered due to the TO device being dead, this does not only affect one delivery, but 19 deliveries. Simply put: the more subscribers, the greater the TO's responsibility.

This bottleneck effect can be especially seen for instant deliveries (i.e. delivery delay lower than 5 seconds). In order to deliver an instant message to subscriber that is not the TO, without any state repairs required, three devices need to be alive at the same time: publishing device, TO device and subscriber device. Since delivery delays are measured from the time where the publishing device is first alive, the probability that the publishing device is alive at this time is 1.0. The probability that the TO device is alive at this time is 1/3 (laptop) or 2/3 (smartphone). The same applies to the subscriber device. Thus, the expected probability that all three devices are alive at the same time, is $(1 * 1/3 * 1/3) = 1/9$ for laptops and $(1 * 2/3 * 2/3) = 4/9$ for smartphones.[12]

The actual results are comparable to this analysis: 6% (laptop) and 38% (smartphone) of all delivery delays are lower or equal to 5 seconds. This can be directly seen in the graphs (Figure 8.13a for laptops and Figure 8.13b for smartphones) (for a detail view on the first 5 seconds, see Figure C.2 in Appendix C).

Additionally, for private instant messaging, one single state repair between the TO device and one single contact's device is enough to deliver any missed notifications. This is not the case for group instant messaging where more than two users are involved – here, multiple state repairs are required.

To see this, consider the following situation: there are three users A, B and C in the group instant messaging scenario. A is the TO. The three application devices are a, b and c respectively. b publishes a new message δ_t while it is dead. When smartphone b becomes alive again, sending becomes possible. Subscriber device c is alive. Thus, the baseline delay $\mathrm{BD}(b,c,\delta_t) = 0$ seconds. b performs a state repair with TO device a. However, a does not notify c about the new data object, since this is not part of the state repair. c does not get notified. Instead, c must itself perform a state repair with TO device a. Since reactive state repairs are used, either a or c must become dead and alive again. Especially for smartphones, this can take a long time due to the long lifetimes and long deadtimes of smartphones.

Delivery delays with TO support device

As with Instant-to-All in private instant messaging, a TO support device helps to improve the delivery delay CDFs: for laptops and *TOSuppServer*, 79.5% of all data objects are delivered in under 3600 seconds (*NoSupp*: 33%). Still, even the improved result of 79.5% is a strong difference to private instant messaging where 90.6% are delivered in under 3600 seconds with laptops and a *TOSuppServer* (see Figure 8.7a and Table 8.3). Until 9000s, the delivery delay CDF for *TOSuppServer* is visibly below the baseline CDF, while for private instant messaging these lines are almost congruent.

[12] $P(A) \cap P(B) = P(A) * P(B)$

The reason for this effect is again the requirement of multiple state repairs (see the discussion above for *NoSupp*). A *TOSuppServer* is always alive and never initiates state repairs. Thus, first the publishing laptop needs to initiate a state repair and at a later point, the subscriber laptop needs to initiate a state repair in order for the instant message to get delivered.

This effect has an even worse impact when smartphones are used as application devices. Here, *TOSuppLaptop* is the most helpful support device for delivery delays over 4000 seconds and even outperforms *TOSuppServer*: this may seem counter-intuitive, but since smartphones rarely switch between lifetimes and deadtimes themselves, a smartphone's best chance for initiating a reactive state repair is a *TOSuppLaptop*, which often switches between deadtimes and lifetimes.

8.7.2 Delays: Offload-First

This section discusses the most relevant differences between Offload-First and Instant-to-All for group instant messaging with regard to delays. If required, differences between group instant messaging compared (Offload-First) and private instant messaging (Offload-First) will also be explained.

Baseline delays for laptops and smartphones

As for Instant-to-All, baseline delays are identical to private instant messaging. To see this, compare the red curves between Figure 8.14a and Figure 8.8a (laptops) / Figure 8.14b and Figure 8.8b (smartphones). Baseline delays are only influenced the lifetime and deadtime distributions of the involved devices, but are independent from the number of subscriber devices or the used Decision Engine.

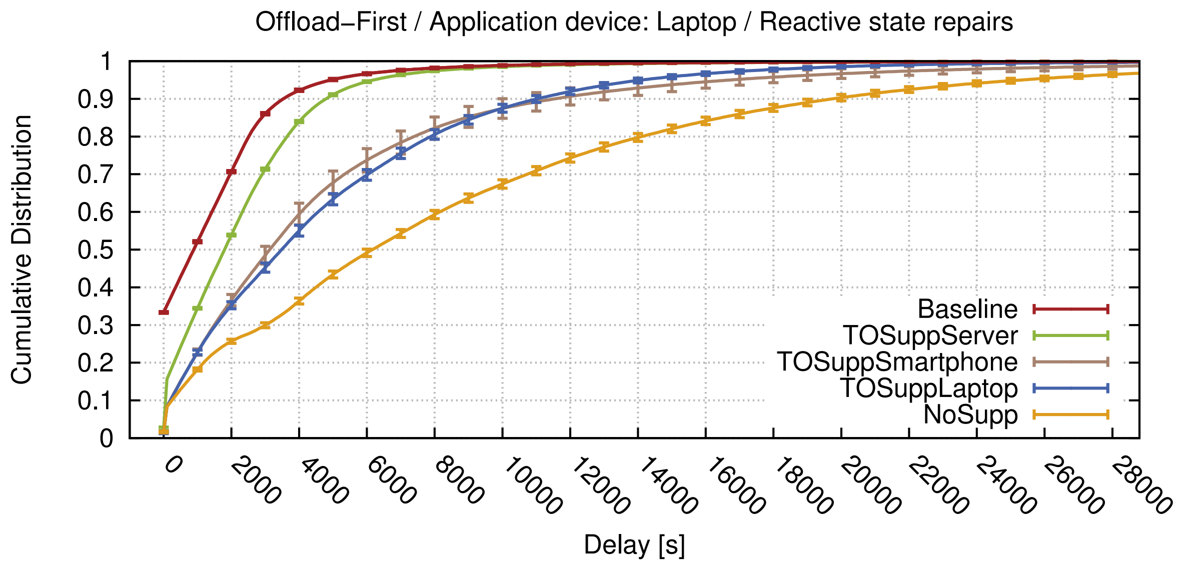
Delivery delays with no support device

For *NoSupp*, the delivery delays between Offload-First and Instant-to-All are very similar. To see this, compare the yellow curves in Figures 8.14 and 8.13. Thus, the same differences between private instant messaging and group instant messaging, that apply to Instant-to-All, also apply here to Offload-First: again, delivery delay CDFs are worse in group instant messaging than in private instant messaging, due to the TO device being a more important bottleneck. Additionally, multiple state repairs are required here. See the explanation in Section 8.7.1 for details.

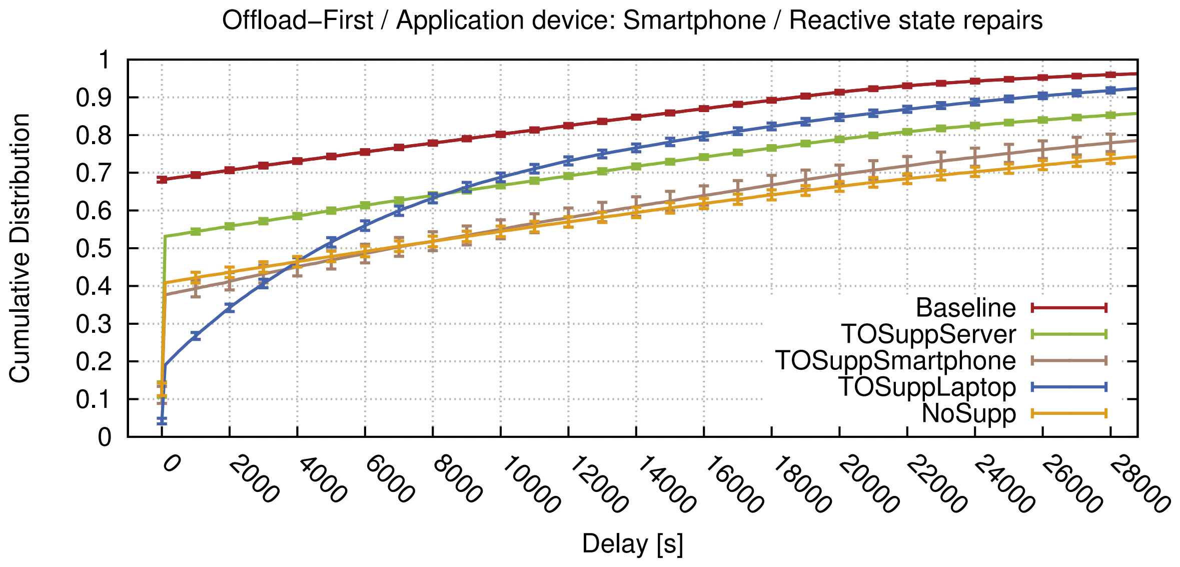
Delivery delays with TO support device

For laptops as application devices, the delivery delays between Offload-First and Instant-to-All are very similar. To see this, compare Figure 8.14a with Figure 8.13a. As for the delivery delays with *NoSupp*, most insights from Instant-to-All can be applied to Offload-First as well – see the explanations in Section 8.7.1 for details.

However, there are notable differences between Offload-First and Instant-to-All if smartphones are used as application devices. With *TOSuppLaptop* and Offload-First only 17% of all instant messages are instantly delivery (i.e. in under 5 seconds), whereas with *TOSuppLaptop* and Instant-to-All 41% are instantly delivered. The reason for this is again *TOSuppLaptop* being the availability



(a)



(b)

Figure 8.14: *Group Instant Messaging: Delay CDFs (Offload-First, reactive states)*

bottleneck: since *TOSuppLaptop* has a lower CCV than the smartphones, each publishing device tries only to reach *TOSuppLaptop*. If *TOSuppLaptop* is unavailable, the whole notification process stalls. In Instant-to-All on the other hand, there is still the TO's smartphone that gets notified and can notify the subscribers even if *TOSuppLaptop* is unavailable. If state repairs are required, *TOSuppLaptop* even outperforms *TOSuppServer* after 8000 seconds, similar as in Instant-to-All – see the explanations in Section 8.7.1 for details.

Another notable difference between Offload-First is the brown curve for *TOSuppSmartphone*: For Instant-to-All, it is similar to *TOSuppServer*, but for Offload-First, it is similar to the curve for no support device, i.e. has no effect. The reason for this difference is the following: with *TOSuppSmartphone*, the TO has two smartphones. For Instant-to-All, the TO can notify the subscribers if at least one of the two smartphones is available. Each smartphone is available with an expected probability of 2/3, thus the probability for at least one of two smartphones being alive is[13]: $(2/3 + 2/3 - 4/9) \approx 0.89$. Thus, for Instant-to-All, a second smartphone achieves an increase of 22% in the overall TO device availability, which is only 11% short of a *TOSuppServer*'s permanent availability. However, for Offload-First a publishing device sends its notification to only one smartphone. Since the application device of the TO is also a smartphone, the additional smartphone is no help.

8.7.3 Delays with Periodic State Repairs

Similar to periodic state repairs for private instant messaging (see Section 8.6.4), devices that rarely switch between lifetime and deadtime gain better delivery delays with periodic state repairs. To see this at the example of Offload-First and smartphones as application devices, compare Figure 8.14b (reactive state repairs) with Figure 8.15b (periodic state repairs = every 500 seconds). With periodic state repairs, the delivery delay CDFs approximate the baseline CDF more closely and faster, as long as a support device is used.

For devices that often switch between lifetimes and deadtimes, periodic state repairs can be harmful to the delivery delays. This can be seen at the example of Offload-First and laptops as application devices, by comparing Figure 8.14a (reactive state repairs) with Figure 8.15a (periodic state repairs = every 500 seconds). For periodic state repairs, the delivery delay CDFs approximate the baseline slower for *NoSupp* or *TOSuppLaptop*.

8.7.4 Costs: Instant-to-All

This section discusses the most notable differences of group instant messaging compared to private instant messaging with regard to costs when using Instant-to-All.

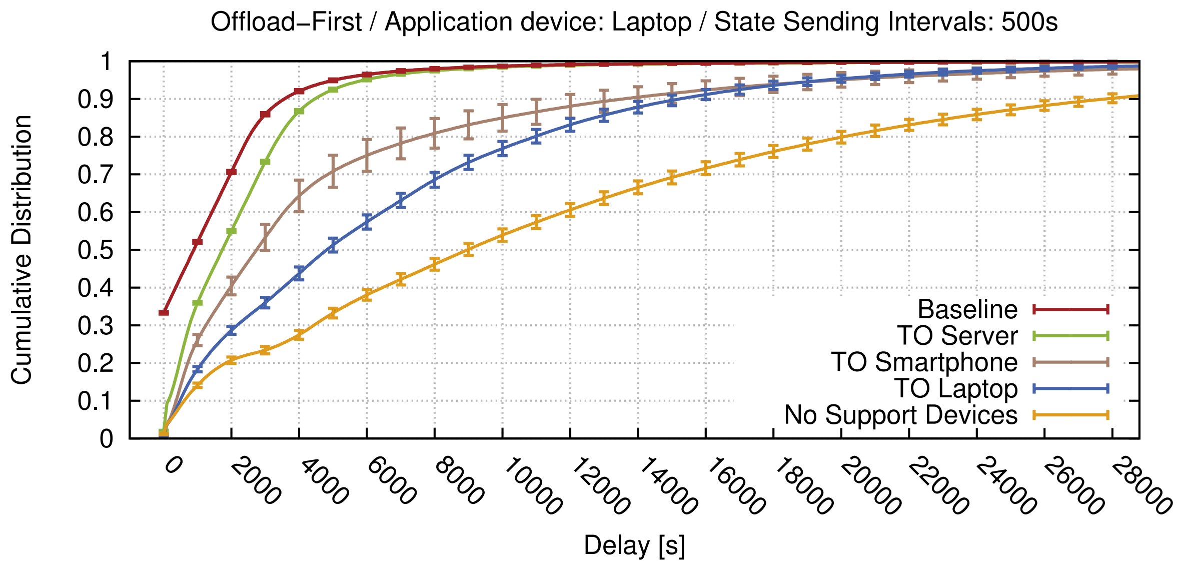
For Instant-to-All, the red-toned bars in Figure 8.16 show the total sent bytes per second, weighed with each device class's penalty value (weighed bytes per seconds or *wb/s*). For each device distribution, the minimum costs (taken from the device with the overall least costs), mean and maximum (taken from the device with the overall highest costs) are shown.

Costs with no support device

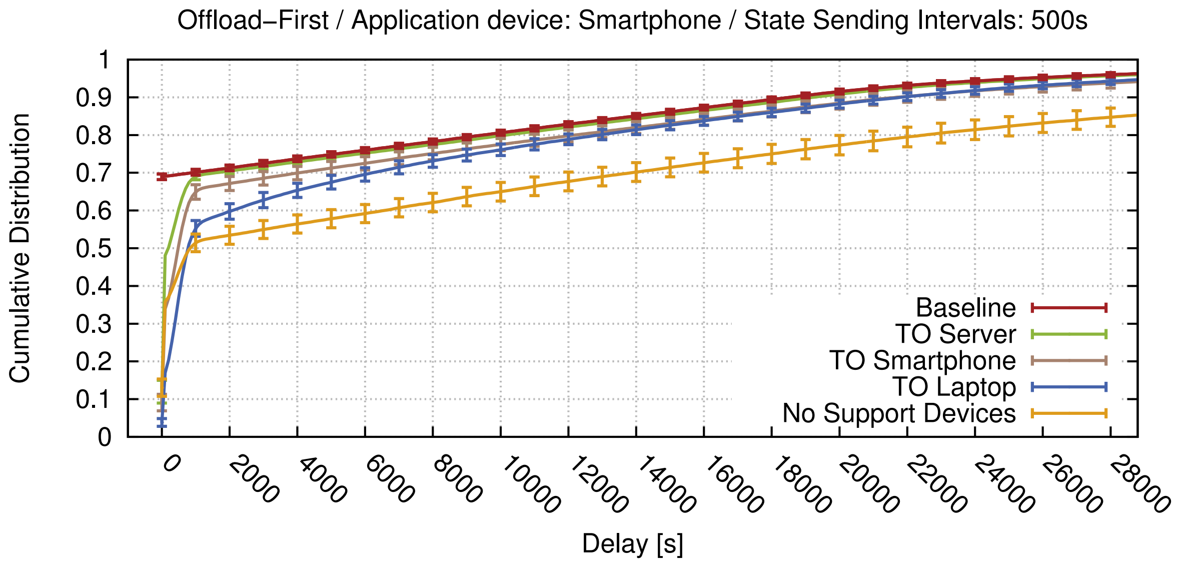
Compared to private instant messaging, minimum, mean and maximum costs rise significantly for group instant messaging with *NoSupp* for both laptops and smartphones as application devices. For laptops in instant group messaging, the mean / maximum costs are 233 / 2400 wb/s. For smartphones, the mean / maximum costs are 293 wb/s / 4273 wb/s.

The reason for the higher mean and maximum costs is the additional effort for the TO device – instead of one delivery there are 19 deliveries per instant message. Additionally, each state repair becomes more expensive for the TO device, since there is only one group instant messaging topic,

[13]Since events A (smartphone 1 is available) and B (smartphone 2 is available) are independent, it is $P(A) \cup P(B) = P(A) + P(B) - P(A) \cap P(B) = P(A) + P(B) - P(A) * P(B)$



(a)

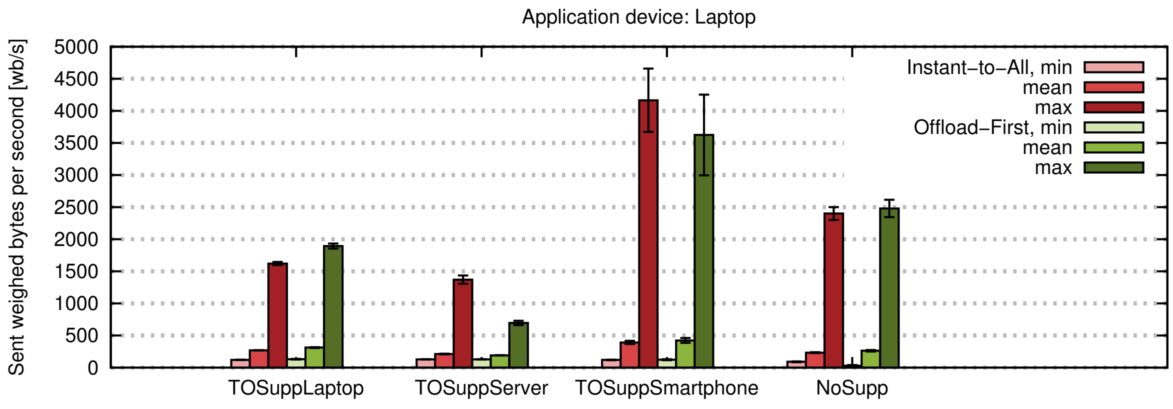


(b)

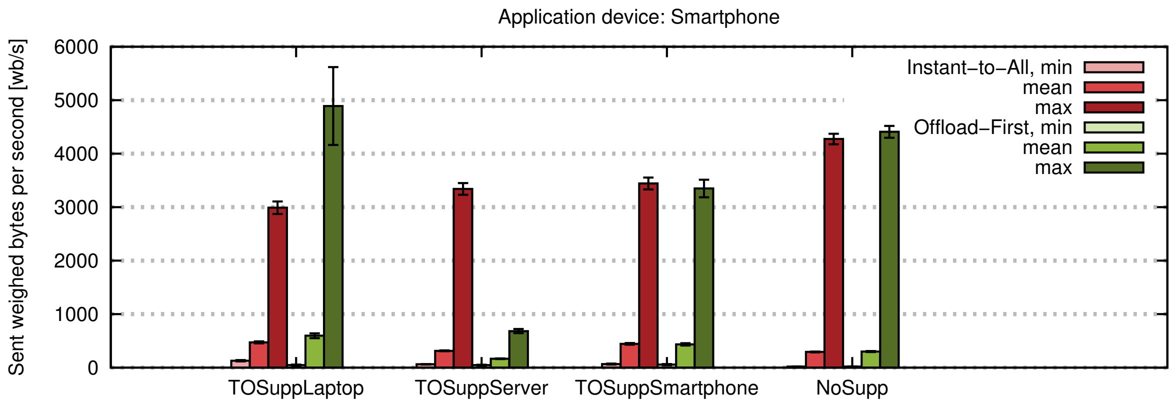
Figure 8.15: *Group Instant Messaging: Delay CDFs (Offload-First, periodic state repairs)*

which is relevant to every device in this scenario. For private instant messaging, each closed group consists of the TO and one distinct contact with their own topic, which is thus not relevant for the other contacts.

The reason for the higher minimum costs are not as intuitive. As discussed earlier in Section 8.6.5, it is a contact's application device which has the minimum cost. However, while a contact device *receives* more instant messages in group instant messaging, it does not *send* more. The only possible explanation is the increased number NOTIFY$_{\text{Rsp}}$ messages that it sends after receiving a notification and during state repair.



(a)



(b)

Figure 8.16: *Group Instant Messaging: Costs*

Costs with TO support device, laptops as application devices

As earlier discussed for private instant messaging, a support device raised the minimum, mean and maximum costs when Instant-to-All was used (see Section 8.6.5). This is different for group instant messaging. As can be seen in Figure 8.16a, mean and maximum are different depending on the choice of the support device:

- *TOSuppLaptop*: here, the maximum is lower than with *NoSupp*, but the mean is higher. This means that the two TO laptops share the effort for state repairs. This is beneficial in group instant messaging: state repairs are more expensive than in private instant messaging since each published data object has 19 other subscribers.

- *TOSuppServer*: here, both the mean and maximum are lower than with *NoSupp*. This means that the *TOSuppServer* can accepts and forward all notifications and is always a target for state repairs (due to its 100% availability) and strongly relieves the TO laptop from the effort for state repairs.

- *TOSuppSmartphone*: here, both the mean and maximum are higher than with *NoSupp*. The reason for this is the higher CCV of a smartphone compared to a laptop's. Due its high availability, the smartphone is often a target for state repairs. The reason for the large confidence interval (98th percentile) was already discussed for private instant messaging (see Section 8.6.5).

Costs with TO support device, smartphones as applications devices

For Instant-to-All and smartphones as application device, a TO support device helps to lower the maximum costs compared to *NoSupp* (see Figure 8.16b), however the mean rises. This indicates that both TO devices share the effort for state repairs. Even a *TOSuppSmartphone* reduces the maximum costs (unlike for laptops), since both the application device and TO support device are smartphones with the same CCV.

8.7.5 Costs: Offload-First

This section discusses the most notable differences of group instant messaging compared to private instant messaging with regard to costs when using Instant-to-All.

For Offload-First, the green-toned bars in Figure 8.16 show the total sent bytes per second, weighed with each device class's penalty value (weighed bytes per second or *wb/s*). Most of the observations for Instant-to-All also apply here. Thus, only the relevant differences shall be discussed here.

Costs with no support device

For smartphones as application devices and *NoSupp*, the maximum costs are only minimal higher than for Instant-to-All. This is a notable difference to private instant messaging, where the costs for Offload-First were almost twice as high as for Instant-to-All. The reason for this was identified as the unneeded notification that gets sent back from the TO device to the publishing device in Offload-First (see Section 8.6.6). This also happens in group instant messaging, however the impact of one additional notification is much lower: 20 instead of 19 notifications are about 5% more costs, whereas in private instant messaging, 2 instead of 1 notification are 100% more costs.

Other than that, the results of Offload-First are comparable to Instant-to-All with *NoSupp* for group instant messaging.

Costs with TO support device

For group instant messaging, both *TOSuppServer* and *TOSuppSmartphone* help to lower the costs compared to *NoSupp*. This applies to laptops and smartphones as application devices. Specifically, the cost reduction for smartphones with *TOSuppServer* is visibly better than for private instant messaging: compared to the 28.7% / 40.3% for mean and maximum in private instant messaging, here 46.9% / 79.6% can be reached (compare Table 8.5 with Table 8.7).

With *TOSuppLaptop*, the mean and maximum costs are higher than for Instant-to-All. The device with the highest costs is the *TOSuppLaptop* (see Table 8.6, which is also applicable for group instant messaging), since all notifications from publishing device are forwarded to it first due to to its lower CCV. Additionally, due to its short lifetimes and deadtimes, it often triggers state repairs. Since multiple state repairs are required in group instant messaging (see Section 8.7.1), this overall results in high sending costs, even if it has a lower CCV than the TO's smartphone.

Table 8.7: *Group instant messaging: Comparison of minimum, mean and maximum costs between Instant-to-All and Offload-First. Application devices are smartphones. Absolute numeric values are in wb/s. Lower values are better.*

	Application Device: Smartphone		
Support Device	**Min Costs (wb/s) Instant-to-All**	**Min Costs (wb/s) Offload-First**	**Costs saved by Offload-First**
TOSuppLaptop	127.9	*46.3*	63.7%
TOSuppServer	61.4	*46.5*	24.2%
TOSuppSmartphone	66.8	*54.9*	17.8%
NoSupp	19.5	*19.0*	2.5%
	Mean Costs (wb/s) Instant-to-All	**Mean Costs (wb/s) Offload-First**	**Costs saved by Offload-First**
TOSuppLaptop	*472.7*	595.6	−25.9%
TOSuppServer	312.7	*166.0*	46.9%
TOSuppSmartphone	443.0	*435.2*	1.8%
NoSupp	*292.8*	300.2	−2.5%
	Max Costs (wb/s) Instant-to-All	**Max Costs (wb/s) Offload-First**	**Costs saved by Offload-First**
TOSuppLaptop	*2990.3*	4890	−63.5%
TOSuppServer	3341	*681.5*	79.6%
TOSuppSmartphone	3450	*3349.1*	2.9%
NoSupp	*4273.5*	4408	−3.1%

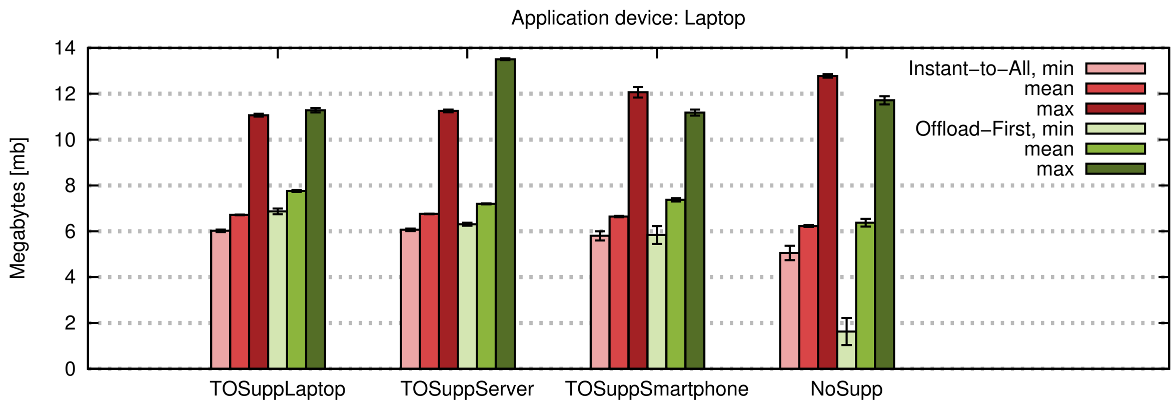
8.7.6 Storage Footprint

In order to estimate the storage footprint for the group instant messaging scenario, Figure 8.17 shows the minimum / mean / maximum size of the Local Data Storage (in unweighed megabytes) after the end of the simulation. Note that the Local Data Storage does not only contain the instant messages, but also all maintenance data objects, such as notifications about new subscribers, new devices, and so on.

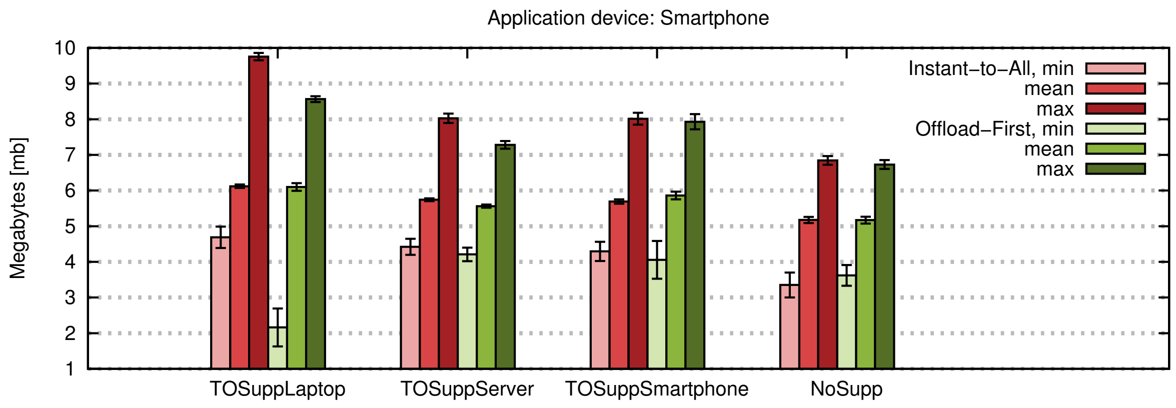
The footprint is visibly higher than for private instant messaging. For the minimum (a contact device) and thus the mean, this is to be expected: each contact does not only receive messages published by the topic owner (as it is the case in private instant messaging), but also from all other group members. Thus, the minimum size as about as large as the maximum size for private instant messaging (see Figure 8.17).

However, the topic owner receives as many instant messages as in the private instant messaging scenario. Still, the maximum size is about twice as high as for private instant messaging. One possible explanation is a higher maintenance overhead due to the higher number of subscribers.

Still, for all device distributions, the final mean (maximum) size is below 7 (14) megabytes. Since 48 hours were simulated, the mean (maximum) would result in less than 105 (210) megabytes per month. Note that is twice the size for maximum, but the tenfold of subscribers compared to private instant messaging.



(a)



(b)

Figure 8.17: *Group instant messaging: final Local Data Storage size*

8.7.7 Key Results

In group instant messaging, there are 20 subscribers instead of 2 per topic. The central role of the TO is even more visible here, since the tenfold of data deliveries depend on him for each published data object. The most important differences to private instant messages are now summarized.

- Data deliveries CDF are considerably worse than for private instant messaging with otherwise identical configurations. The reason for this is the central role of the TO: if all TO devices are dead, not only one but 19 delivieries cannot be performed. Since the CDF takes all deliveries into account, this gives visibly worse results.

- Thus, unlike for private instant messaging, scalability is a problem with a rising number of subscribers for a given topic.
- If contact device c_1 wants to forward a published data object to the TO, but all TO devices are dead at that time, two state repairs are required for each subscriber: the first between c_1 and the TO device, the second between the TO device and the subscriber device
- The multiple state repairs have an especially negative effect on the delivery delay CDFs of smartphones where reactive state repairs are rare. Periodic state repairs are beneficial here.
- Costs are overall higher for all devices.
- Higher costs apply especially to the TO devices: for each published data object, not only one notification, but instead 19 must be sent.
- Higher costs also apply to the contacts: they do not send more, but receive more data objects, thus they have to send more $\text{NOTIFY}_{\text{Rsp}}$ messages
- Proportionally, Offload-First can save even more costs over Instant-to-All than in private instant messaging. For the mean / maximum cost, up to 46.9% / 79.6% of the costs can be saved (smartphones as application devices).

8.8 Summary

This chapter presented the evaluation of SODESSON and SocioPath, based on an implementation in the overlay simulation framework OverSim. The extensions to OverSim were described first.

The major part of this chapter discussed two different application use-cases: private instant messaging and group instant messaging. For both use-cases, different device distributions (laptops and smartphone as devices which run the application, plus different support devices for the topic owner) and the Decision Engines Instant-to-All and Offload-First were regarded. The main performance metrics for the evaluations are delivery delays and sending costs. The former are compared to a theoretical lower bound, the baseline delay. The latter are weighed with a cost penalty factor, depending on the respective device classes and their capability. The key results for private instant messaging were summarized in Section 8.6.8, whereas the key results for group instant messaging are in Section 8.7.7.

As bottom line, the evaluations have shown that SocioPath is a functional approach to the presented use-cases. Specifically state repairs help to deliver instant messages after intermittent device deadtimes and thus approximate the delivery delays very closely to the baseline delays. Specifically support devices can support a faster approximation: for private instant messaging, with a home server as a TO support device, the difference between delivery delay and baseline delay is always less than 1%. Depending on the expected lifetimes and deadtimes of the devices, periodic state repairs are more preferrable than reactive state repair.

However, SocioPath's performance is strongly dependent on the TO devices. Since the topic owner has a central position in every closed group, no data objects can be delivered without his device. Additionally, topic owner devices have the majority of costs. The central responsibility of the topic owner becomes even greater when the number of subscribers for one topic grows (i.e. in group instant messaging) and even more deliveries depend on him. Also, the costs grow for group instant messaging.

It is possible to reduce the costs with Offload-First compared to Instant-to-All, if a support device is used: for private instant messaging, up to 40.3% can be saved, whereas for group instant messaging even 79.6% can be saved. However, a TO device with low costs and low availability can become an availability bottleneck in Offload-First.

The mean storage footprint for the Local Data Storage is less than 15 megabytes per month for private instant messaging, and less than 105 megabytes per month group instant messaging. This includes not only the published data objects by the application, but also SocioPath's maintenance data objects.

Chapter 9

Conclusion and Perspectives

Private user-to-user communication in a closed group (*U2U communication*) is a key communication scenario of our everyday lives. In order to store and deliver data objects to the users' personal devices, typically third-party application providers are involved. They relieve the data storage and data object delivery effort from the personal devices of the closed group members and thus help raising the data availability of each published data object. However, since these data objects are only meant for a closed group, the third-party provider must also perform access control. Here, *explicit application metadata* is required for the provider to identify and address the correct users. Typically, a unique identifier for each user is a part of explicit metadata in U2U communication. Examples are email addresses or online social network profiles. As soon as users are addressed by unique identifiers, the third-party provider becomes a threat to the users' privacy since it can track their service usage behaviour. In combination with *implicit application metadata*, the provider can further learn about its users – for example, identify very close social contacts by the number of interactions or learn about specific interests by the user's data object consumption patterns. Depending on the user identifiers, even a user's real-world identity can be revealed. While there exist privacy enhancing techniques that enable third-party anonymity for the users, these are mostly designed for obfuscating connection metadata in the network.

For this reason, this thesis proposed and investigated User-Centric Networking as an alternative to third-party providers for U2U communication. User-Centric Networking is based on *self-sufficiency*, which is the idea that delivering data objects from the publisher to the subscriber shall be handled by the personal devices of the closed group members only. The main advantage of this approach is an improvement of the closed group members' privacy, since application metadata is not visible to a third party. There exists related work on approaches where data objects are only stored on trusted personal devices. However, the assumed trust in those approaches is always of a general kind in specific users such as friends or family. Trust on a per-data-object basis was not regarded so far and makes User-Centric Networking a novel approach to this problem.

Leveraging personal devices has disadvantages, as they are heterogenous with regard to device availability and device capability. User-Centric Networking follows two design goals that take such heterogeneity into account: first, *partition tolerance* shall handle intermittent lack of availability between devices and let them recover from missed data object deliveries. Second, *resource awareness* shall take the individual devices' availabilities and devices' capabilities into account for the policy of forwarding data objects from the publisher to the subscribers. In this context, three mutually exclusive goals were identified: low delivery delays, low costs and high privacy. Depending on the distribution of devices and privacy demands within the closed group, one forwarding policy can be more preferable than another. A technical solution must offer flexibility here.

This thesis has made the following central contributions:

- Design of a U2U communication model based on the notion of topic owners
- Definition of the novel User-Centric Networking paradigm and specification of its design goals
  - *Self-sufficiency:* U2U communication within a UCN with the group members' personal devices
  - *Partition tolerance*: coping with personal devices' intermittent unavailability
  - *Resource-awareness*: taking different capabilities of personal devices into account
- Technical realization of User-Centric Networking
  - *SODESSON*: a middleware for U2U communication, based upon the designed U2U communication model
  - *SocioPath*: a self-sufficient and partition-tolerant Data Distribution Protocol (DDP) for the SODESSON middleware, which is compliant to the design goals of User-Centric Networking
  - *Instant-to-All, Offload-First, Helping-Friends*: three Decision Engines for SocioPath which implement different forwarding policies and demonstrate SocioPath's flexibility with regard to resource awareness

9.1 Results of This Thesis

Design of a U2U communication model based on the notion of topic owners

As a first result, a model for U2U communication scenario was created. A published *data object* with a specific *topic* shall be shared between the members of a *closed group* only. Before any data object can be published, the *topic owner* defines from his *contacts* which users are allowed publishers and allowed subscribers for a given topic. The applicability of this model was shown for three third-party provider schemes: Centralized Service Provider, Federated Service Providers and structured P2P overlay networks.

For this model, required trust dependencies between the closed group and any third-party providers were analyzed. Four important security and privacy goals were analyzed: content data confidentiality, content data integrity, content data availability and application metadata privacy. The three third-party provider schemes were analyzed with regard to these goals. As one result of the trust dependency considerations, all closed group members must blindly trust a third-party provider with regard to application metadata privacy if unique identifiers are used. This result supported the motivation for User-Centric Network.

Definition of User-Centric Networking and specification of its design goals

User-Centric Networking was proposed as a novel alternative to third-party provider schemes for U2U communication. U2U communication in User-Centric Networking is *self-sufficient*, i.e. data object delivery from the publisher to the subscribers is handled by the personal devices of the closed group members only – not even by devices of generally trusted contacts (e.g. friends, family or co-workers) outside of the closed group. This is a new concept compared to the state-of-the-art. The main disadvantage is that all efforts for data object storage and delivery cannot be outsourced to a third party, but need to be handled by the personal devices of the closed group members. These personal devices are heterogeneous with regard to two aspects: *device availability* and *device capability*.

The next results are two proposed solutions for the problem of device heterogeneity: first, *partition tolerance* shall cope with intermittent device availability: if a network splits into two partitions $\mathcal{A}$ and $\mathcal{B}$, it is still possible for devices in each respective partition to publish and deliver data objects as long as there is at least one topic owner device in the same partition. Additionally, if two partitions merge at a later time, the devices in $\mathcal{A}$ shall get data objects delivered that were published earlier in B and vice-versa. The second design goal is *resource awareness*, which shall take devices' availabilities and devices' capabilities into account and enable different forwarding policies for data objects based on the given resources. The impact of different forwarding policies was explained with the help of three examples.

Based on the comparison of third-party provider schemes and User-Centric Networking with regard to trust dependencies, a privacy model with four different levels was created. Each provider scheme, whether third-party provider schemes or self-sufficient forwarding policies can be mapped into this model. This enables the comparison of different provider schemes with regard to privacy.

Technical realization of User-Centric Networking

The key contribution of this thesis is the technical realization of the presented concepts and design goals. This technical realization consists of three major parts: SODESSON, which is complemented by SocioPath, which in turn is complemented by a Decision Engine.

SODESSON is a middleware for enabling U2U communication applications and implements the U2U communication model presented in the beginning. This means, SODESSON enables applications to publish data objects and to subscribe to topics. Besides topics, SODESSON complies with the U2U entities of topic owners, allowed publishers and allowed subscribers. It features an application interface which builds upon these entities. Additionally, the interface supports decoupling of notifications and data object retrieval. This decoupling plays an important role for establishing resource awareness in User-Centric Networking later. SODESSON by itself is agnostic of any device handling and provider schemes. It can leverage third-party provider schemes as well as User-Centric Networking. To this end, SODESSON needs to be complemented with a Data Distribution Protocol (DDP) which handles the inter-device communication.

SocioPath is a DDP for SODESSON which realizes the novel concept of User-Centric Networking. SocioPath's main features are realization of the design goals in User-Centric Networking: self-sufficiency, partition tolerance and resource awareness. As a basis for self-sufficiency, each device holds a list of all devices of all contacts and thus can communicate directly with them. Therefore, each user can take the role of a topic owner and define allowed publishers and allowed subscribers from SODESSON's Contact List. A publishing device can forward a data object to a topic owner's device (since the topic owner is a contact of the publisher) and the topic owner's device in turn can deliver data objects to the devices of the subscribers, which are a subset of his contacts. This delivery can be decoupled into notifications and the actual data object content, giving each device its own decision if and when to retrieve a data object.

SocioPath achieves partition tolerance by so-called *state repairs*. A state repair is a process of set reconciliation between two devices with regard to the notifications that they respectively store. Depending on the users of the regarded devices and their roles (publisher, subscriber, topic owner), different notifications need to be synchronized between two devices. These notification are identified by their topic, i.e. there are different relevant topics depending on the constellation of the affected users' roles. The respective notifications are hashed into a space-efficient Bloom Filter. For additional space efficiency, each device remembers the timestamp of each last successful state repair for all other known devices. Any notifications older than that timestamp can be disregarded for the next state repair and do not need to be hashed into the Bloom Filter.

SocioPath is designed to support different forwarding policies for data objects. Depending on the available resources, one policy can be more preferable than another. Factors here are for example the distribution of personal devices with regard to number, device availabilities and device capabilities or different demands on privacy. In order to be flexible, SocioPath defines different points in its protocol cycle which are left to be defined by a so-called *Decision Engine (DE)*. First, a workflow was presented with well-defined DE events and DE actions: for each DE event, a DE must define to react to this event. Each DE action involves sending messages to other devices – here the DE must select a suitable subset of the targeted devices. Each Decision Engine takes this workflow as a template, and specifies its behaviour on these well-defined events and actions. To establish this concept, three different DEs *Instant-to-All, Offload-First* and *Helping-Friends* were presented. Each DE integrates in SocioPath and aims at a different trade-off: Instant-to-All aims at low delivery delays and high privacy. Offload-First aims at low costs and high privacy. Helping-Friends aims at low delivery delays and low costs.

Finally, SODESSON and SocioPath were implemented for and evaluated with the help of the overlay network simulator OverSim [7]. The evaluation metrics were delivery delays and sending costs. Different device distributions were regarded, where each user either has a smartphone (high availability, high costs) or a laptop (low availability, medium costs). Optionally, the topic owner has an additional support device, either an additional smartphone, laptop or home server (high availability, low costs). Results have shown that it is possible to gain delivery delays that closely approximate to a baseline, which is a theoretical lower bound that depends on the involved device's individual availabilities. This result applies to instant delivery (low delays of a few milliseconds) as well as later delivery via a state repair. For private instant messaging, with a home server as a TO support device, the difference between delivery delay and baseline delay is always less than 1%.

If a device rarely changes its availability status, periodic state repairs are to be preferred over reactive state repairs. In terms of delay, SocioPath is applicable for private instant messaging and group instant messaging under the assumed model. However, if Offload-First is used, the topic owner's device can become an availability bottleneck and increase delivery delays. For costs, each device class's abstract CCV (smartphone = 10, laptop = 5, home server = 1) is multiplied as a penalty with each sent byte. Under this model, Offload-First lowers the average minimum/mean/maximum costs up to 63.7%/46.9%/79.6% (smartphones as application devices, group instant messaging). Thus, the selection of a suitable Decision Engine has a notable impact, depending on the respective scenario.

9.2 Perspectives

This thesis has laid the groundwork for User-Centric Networking, a novel approach to U2U communication. Evaluation results have shown promising results that can be further evaluated, based on the concepts realized in SODESSON and SocioPath, as well as the existing implementation in OverSim. Specifically, the Decision Engine system in SocioPath allows for pursuing new ideas and improvements.

Simulative evaluations have shown that there is not one optimal solution for a forwarding policy, depending on the distribution of devices among the users and on which of the three goals in User-Centric Networking the communication scenario focuses: low delivery delays, low costs or high privacy. Instead, flexible policies are required with changing scenarios. With its Decision Engine system, SocioPath offers a solution to designing new policies.

As a perspective, new and more sophisticated Decision Engines can be developed for SocioPath. For example, instead of the abstract CCV used for costs in this thesis, more detailed measurements of device resources can be made (with regard to both availability and capability) and be taken into account. Additionally, user behaviour is an interesting source for Decision Engines, e.g. by taking into account which user uses which application on which device at a specific time of the day and adapt the forwarding polices accordingly.

An important issue in User-Centric Networking is the centrality of the topic owner. While subscribers can help delivering data objects to other subscribers (e.g. with the Helping-Friends DE), all access control is performed on the topic owner's devices only. If these are unavailable, it is not possible to subscribe to a new topic of that owner and notifications about newly published data objects cannot be sent to subscribers even if their devices are available. One possible solution here could be the relaxation of the topic owner role. For example, there could be multiple users equally assuming the role of the topic owner. Additionally, with the current design it is not possible to pass ownerships from one user to another.

In terms of privacy, only application metadata was regarded in this thesis. For example, an attacker in the physical network can currently observe devices of contacts that communicate with each other. For ensuring communication metadata privacy, SocioPath could be combined with existing anonymization approaches, e.g. as mix networks such as *Tor* [25].

Appendix A

SODESSON Application Interface: Publish/Subscribe

- updateTopic App ► Middleware
  Set allowed publishers and allowed subscribers for a topic the current user is topic owner of. Create topic if it does not exist yet. Delete topic if allowed publisher and allowed subscriber sets are empty.

  Arguments:

  – title: `String`
    Topic title, as defined in Section 4.3.1.

  – allowedPublishers: `Set<Contact>`
    Set of Contact List entries that are allowed to publish data objects to this topic.

  – allowedSubscribers: `Set<Contact>`
    Set of Contact List entries that are allowed to subscribe to this topic.

  – privacyLevel: `Enum`
    (Optional) Demanded privacy level (1-4) as defined in Section 3.6.2. This has to be supported by the DDP, otherwise this parameter has no effect.

- publish App ► Middleware
  Publish a data object.

  Arguments:

  – topic: `Topic`
    Topic as defined in Section 4.3.1

  – OID: `Unsigned Integer`
    (Optional) OID as defined in Section 4.3.1. Set only if the application wants to set a specific value. If empty, the middleware sets a value, as defined in section 4.3.1.

  - content: BinaryData
    The content of the data object to be published. Either passed directly in this field or as filepath reference (see argument "inline")
  - inline: Boolean
    True: The data object is directly passed from the application to SODESSON in the argument "content".
    False: The data object is a file on the local device's filesystem and the filepath is passed to SODESSON.
  - TTL: Unsigned Integer
    Time-to-live in seconds for the data object until it is seen obsolete and can be safely deleted / ignored. Counted from the time of this publish call. 0 = forever.

  Return value: Unsigned Integer
  The *OID* is returned. If a non-empty *OID* different was provided, exactly that *OID* is returned. Otherwise the *OID* chosen by SODESSON is returned.

- subscribe App ► Middleware
  Subscribe to a topic. SODESSON makes a best effort to deliver a subscription request to the device(s) which are responsible for handling subscriptions to that topic. Until SODESSON receives a subscription grant / denial, the request is pending from the application point of view.

  Arguments:

  - topic: Topic
    Topic as defined in Section 4.3.1
  - handle: 32-bit unsigned integer
    Handle to identify the subscribing application. A handle is a unique identifier for an application running on this device. By passing it here, SODESSON learns which application(s) are subscribed to a specific topic and need to be notified on an incoming data object. This way, multiple applications can subscribe to the same topic.

  Return value: Boolean
  True: A storage device responsible for enforcing access control application has acknowledged that the requesting user is an allowed subscriber and successfully entered him as a subscriber.
  False: A storage device responsible for enforcing access control application has denied that the requesting user is an allowed subscriber.

  Note: Until the subscribe request receives this response, the application is in a state where the subscription request is pending. If a responsible storage device is unavailable, the pending status persists in the meantime.

  Note: An application is free to repeat a subscription request.

- unsubscribe App ► Middleware
  Unsubscribe from a topic. SODESSON makes a best effort to deliver a subscription request to the device(s) which are responsible for handling subscriptions to that topic. Until SODESSON receives a unsubscription acknowledgement, any incoming data objects tied to the given topic are not passed to the application anymore.

  Arguments:

  - topic: Topic
    Topic as defined in Section 4.3.1
  - handle: 32-bit unsigned integer
    Handle to identify the subscribing application. A handle is a unique identifier for an application running on this device. By passing it here, SODESSON knows which application wants to unsubscribe.

  Return value: Boolean
  True: The application was subscribed and SODESSON processes the unsubscription request.
  False: The application has already unsubscribed or was not subscribed in the first place.

- registerApp App ► Middleware
  Register an application.

  Return value: 32-bit unsigned integer
  A handle, which identifies the application in further requests.

- unregisterApp App ► Middleware
  Unregister an application via its handle.

  Arguments:

  - handle: 32-bit unsigned integer
    The handle to be unregistered.

  Return value: Boolean
  True: The application was successfully unregistered.
  False: The application is already unregistered or handle is invalid.

- notifyApp Middleware ► App
  A data object was published on another device and this device has received notice about it. Notify an application, which is subscribed to the data object's topic, about the data object. Since a DDP might decouple a **notification** about the publication of a data object and the actual data object transfer, the data object's content might not already be on this device.

  If argument "content" is empty, this method call is a pure notification about the data object, without the data object's content being on the device. The data object's actual transfer can be triggered by the application via the method retrieve. It is then delivered to the application via another notifyApp call in the future, with a non-empty "content" argument.

  Arguments:

- topic: Topic
  Topic as defined in Section 4.3.1
- OID: 64-bit unsigned integer
  OID as defined in Section 4.3.1. Originally set on the publishing device, either by the application or SODESSON middleware (see method publish).
- timestamp: 32-bit unsigned integer
  Data object timestamp as defined in Section 4.3.1. Originally set on the publishing device by the SODESSON middleware.
- content: BinaryData
  (Optional) If not empty: The content of the published data object. Either passed directly in this field or as filepath reference (see argument "inline").
  If empty: This method call is a pure notification about the data object.
- inline: Boolean
  True: The data object is directly passed to the application from SODESSON in the argument "content".
  False: Argument "content" is empty or the data object is a file on the local device's filesystem and the filepath is passed to the application.

• getNotifyApps App ► Middleware
An application calls this method to make SODESSON repeat notifyApp calls for a given topic. SODESSON looks up all entries in the Local Data Storage and replys with a notifyApp for each entry. This method is helpful for newly installed or reinstalled application to gain knowledge about existing data objects in the Local Data Storage.

Arguments:

- topic: Topic
  Topic as defined in Section 4.3.1

• retrieve App ► Middleware
Trigger the download of a data object the application was earlier notified about via notifyApp with an empty "content" argument. It is transparent to the application if the object is already available on the same device or if it has to be downloaded by SODESSON from another device first. On success, this results in another notifyApp call with an non-empty "content" argument.

Arguments:

- topic: Topic
  Topic as defined in Section 4.3.1
- OID: 64-bit unsigned integer
  (Optional) OID as defined in Section 4.3.1. Originally set on the publishing device, either by the application or SODESSON middleware (see method publish).

– timestamp: 32-bit unsigned integer
  Data object timestamp as defined in Section 4.3.1. Originally set on the publishing device by the SODESSON middleware.

Appendix B

SODESSON Application Interface: Contact Management

- `registerUser` App ► Middleware
  Register a new user with this device being the first one in his device pool. SODESSON creates a fresh public / private keypair for this user and initializes empty data structures.

  Arguments:

  - keypairFilepath: `String`
    This path to a non-existing file tells SODESSON where to write the public/private key file. If the file exists it is not overwritten, rather an error code is returned.

  Return value: `Enum`
  The method returns one of the following status codes:

  (0) Success

  (1) Write error while writing private key

  (2) Error while generating private key

- `importUser` App ► Middleware
  Register an existing user on another device, i.e. adding this device to the user's device pool. User requires his public / private keypair from another device for successful import.

  Arguments:

  - keypairFilepath: `String`
    This path to a file tells SODESSON where to find the existing user's public/private key files.

  Return value: `Enum`
  The method returns one of the following status codes:

  (0) Success

  (1) Invalid keypair at given filepath

- getContacts — App ► Middleware
  Return the contents of the Contact List to the requesting application.

  Return value: Set<UID, String>
  List of <User ID, User Alias> tuples

- addContact — App ► Middleware
  Add a contact to the Contact List.

  Arguments:

  - publicKeyFilepath: String
    This path to a file tells SODESSON where to find the contact's public key file.
  - alias: String
    (Optional) User alias for better human readability. Can be freely defined by the device owner.

- removeContact — App ► Middleware
  Remove a contact from the Contact List.

  Arguments:

  - userId: UID
    Contact's UID as defined in Section 4.3.1.

- editContactAlias — App ► Middleware
  Edit the alias of an existing contact.

  Arguments:

  - userId: UID
    Contact's UID as defined in Section 4.3.1.
  - alias: String
    User alias for better human readability.

Appendix C

SocioPath – Additional Evaluation Results

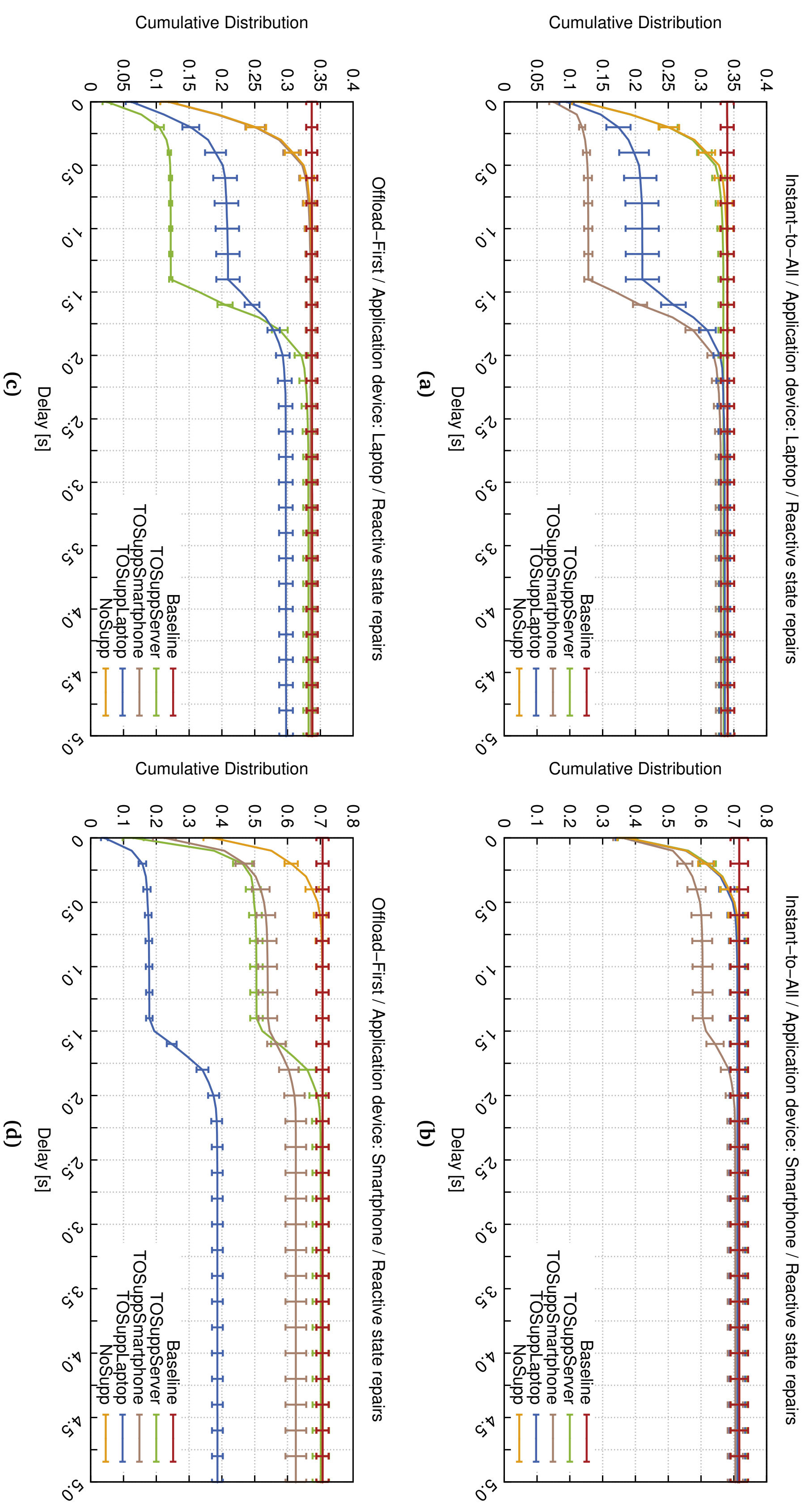
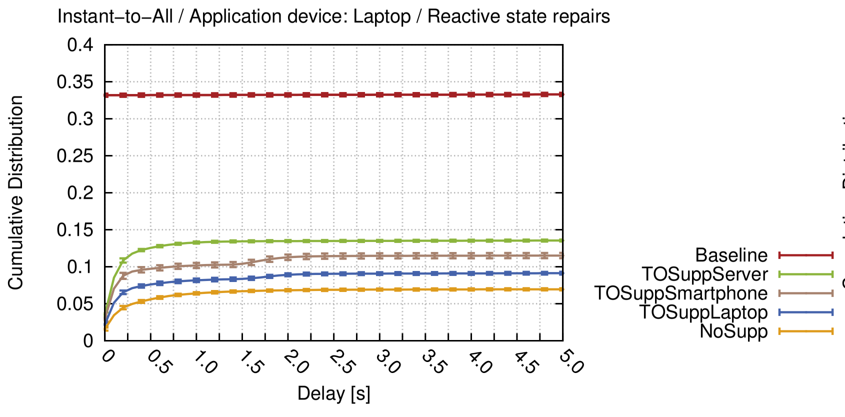
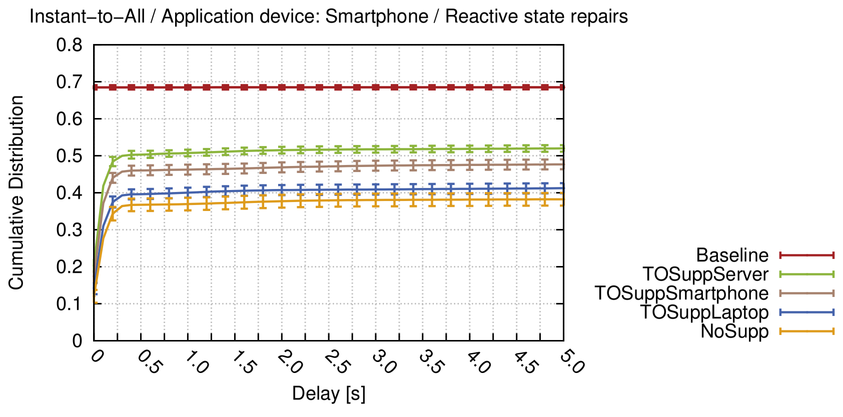


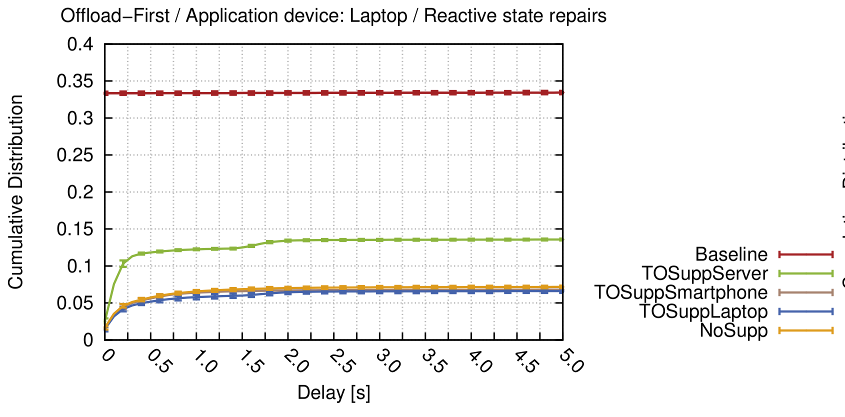
Figure C.1: *Private Instant Messaging: Delay CDFs (detail view for the first 5 seconds)*



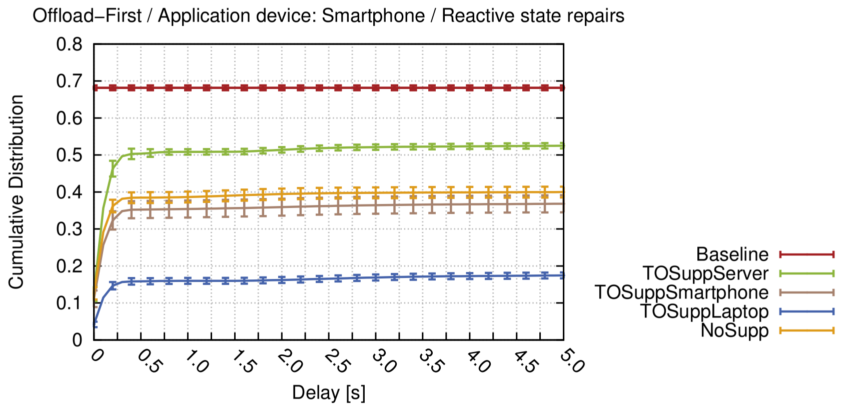
(a)



(b)



(c)



(d)

Figure C.2: *Group Instant Messaging: Delay CDFs (detail view for the first 5 seconds)*

Appendix C

Bibliography

[1] L. M. Aiello and G. Ruffo. “LotusNet: Tunable Privacy for Distributed Online Social Network Services.” In: *Computer Communications* 35.1 (2012), pp. 75–88.

[2] L. M. Aiello et al. “An Identity-Based Approach to Secure P2P Applications with Likir.” In: *Peer-to-Peer Networking and Applications* 4.4 (Jan. 2011), pp. 420–438.

[3] 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.

[4] R. Baden et al. “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.

[5] M. Bastian, S. Heymann, and M. Jacomy. “Gephi: An Open Source Software for Exploring and Manipulating Networks.” In: *Proceedings of the Third International AAAI Conference on Weblogs and Social Media*. San Jose, CA, USA, 2009.

[6] I. Baumgart and F. Hartmann. “Towards Secure User-Centric Networking: Service-Oriented and Decentralized Social Networks.” In: *2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops*. Ieee, Oct. 2011, pp. 3–8.

[7] 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.

[8] I. Baumgart, B. Heep, and S. Krause. “OverSim: A Scalable and Flexible Overlay Framework for Simulation and Real Network Applications.” In: *9th International Conference on Peer-to-Peer Computing (IEEE P2P’09)*. Sept. 2009, pp. 87–88.

[9] I. Baumgart and S. Mies. “S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing.” In: *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS ’07), Hsinchu, Taiwan*. 2007.

[10] K. Bennett et al. *GNUnet - A truly anonymous networking infrastructure*. Tech. rep. 2002.

[11] A. Bielenberg et al. "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.

[12] *bloom*. URL: `https://github.com/arashpartow/bloom` (visited on 05/11/2016).

[13] B. H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors." In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426.

[14] E. Brewer. "Towards Robust Distributed Systems." In: *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*. 2000, pp. 7–10.

[15] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey." In: *Internet Mathematics* 1.4 (Jan. 2004), pp. 485–509.

[16] S. Buchegger et al. "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.

[17] K. C. Claffy. "CAIDA: Visualizing the Internet." In: *Internet Computing Online* (2001), p. 88.

[18] I. Clarke et al. *Private Communication Through a Network of Trusted Connections: The Dark Freenet*. Tech. rep. 2010.

[19] B. Cohen. "Incentives Build Robustness in BitTorrent." In: *Workshop on Economics of Peer-to-Peer systems*. Vol. 6. Berkeley, CA, USA, June 2003, pp. 68–72.

[20] J. Crowcroft et al. "Unclouded Vision." In: *Distributed Computing and Networking*. Ed. by M. K. Aguilera et al. Vol. 6522. Lecture Notes in Computer Science. Springer, 2011, pp. 29–40.

[21] L. A. Cutillo, R. Molva, and T. Strufe. "Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life Trust." In: *IEEE Communications Magazine*. Consumer Communications and Networking 47.12 (Dec. 2009), pp. 94–101.

[22] G. Danezis et al. *Privacy and Data Protection by Design – from Policy to Engineering*. Tech. rep. December. European Union Agency for Network and Information Security (ENISA), Dec. 2014. arXiv: `1501.0372`.

[23] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. IETF RFC 5246. Aug. 2008.

[24] R. Dingledine and N. Mathewson. "Anonymity Loves Company: Usability and the Network Effect." In: *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006)*. Ed. by R. Anderson. Cambridge, UK, 2006.

[25] 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.

[26] S. Dohrmann and C. M. Ellison. “Public-Key Support for Collaborative Groups.” In: *1st Annual PKI Research Workshop*. Gaithersburg, Maryland, USA, 2002, pp. 139–148.

[27] P. Dorling. *Snowden reveals Australia’s links to US spy web*. Sydney, Australia, July 2013. URL: `http://www.smh.com.au/world/snowden-reveals-australias-links-to-us-spy-web-20130708-2plyg.html`.

[28] J. R. Douceur. “The Sybil Attack.” In: *IPTPS ’02: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag, 2002, pp. 251–260.

[29] *Dropbox*. URL: `https://dropbox.com` (visited on 05/11/2016).

[30] M. Dürr, M. Maier, and F. Dorfmeister. “Vegas - A Secure and Privacy-Preserving Peer-to-Peer Online Social Network.” In: *Proceedings of 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust and 2012 ASE/IEEE International Conference on Social Computing (PASSAT/SocialCom 2012)*. Amsterdam, The Netherlands: IEEE, Sept. 2012, pp. 868–874.

[31] J. Dwyer. “Four Nerds and a Cry to Arms Against Facebook.” In: *New York Times* (May 2010), A19.

[32] Electronic Frontier Foundation (EFF). *Why Metadata Matters*. URL: `https://www.eff.org/de/deeplinks/2013/06/why-metadata-matters` (visited on 10/26/2015).

[33] *Facebook*. URL: `https://facebook.com` (visited on 05/11/2016).

[34] *Facebook: Company Info*. URL: `http://newsroom.fb.com/company-info/` (visited on 11/22/2015).

[35] A. Famulari and A. Hecker. “Mantle: A Novel DOSN Leveraging Free Storage and Local Software.” In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7593 LNCS (2013), pp. 213–224.

[36] J. Farina, M. Scanlon, and M.-T. Kechadi. “BitTorrent Sync: First Impressions and Digital Forensic Implications.” In: *Digital Investigation* 11 (May 2014), S77–S86.

[37] Financial Times. *Snapchat triples video traffic as it closes the gap with Facebook*. URL: `http://www.ft.com/cms/s/0%7B%5C%%7D252Fa48ca1fc-84e7-11e5-8095-ed1a37d1e096.html` (visited on 11/22/2015).

[38] 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.

[39] A.-T. Gai and L. Viennot. “Broose: a Practical Distributed Hashtable based on the De-Bruijn Topology.” In: *Fourth International Conference on Peer-to-Peer Computing (P2P 2004)*. Zurich, Switzerland, Aug. 2004, pp. 167–174.

[40] S. Gilbert and N. Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.” In: *ACM SIGACT News* 33.2 (June 2002), p. 51.

[41] GlobalWebIndex. *Device Summary Q1 2015*. Tech. rep. 2015, pp. 1–8.

[42] Google. *Gmail now has over 900M users!* 2015. URL: `https://plus.google.com/+Gmail/posts/AjktcDswdKh` (visited on 11/25/2015).

[43] *Google+*. (Visited on 08/06/2015).

[44] *Google Docs*. URL: `https://docs.google.com` (visited on 05/11/2016).

[45] K. Graffi et al. "LifeSocial.KOM: A Secure and P2P-based Solution for Online Social Networks." In: *Proceedings of the 8th Annual IEEE Communications and Networking Conference*. Jan. 2011, pp. 554–558.

[46] D. Gray and D. Citron. "The Right to Quantitative Privacy." In: *Minnesota Law Review* 98 (2013), pp. 62–144.

[47] J. Gray and D. Siewiorek. "High-Availability Computer Systems." In: *Computer* 24.9 (Sept. 1991), pp. 39–48.

[48] J. Gray and C. van Ingen. *Empirical Measurements of Disk Failure Rates and Error Rates*. Tech. rep. December. Microsoft Research, 2005, pp. 1–3.

[49] 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.

[50] S. Guha, K. Tang, and P. Francis. "NOYB: Privacy in Online Social Networks." In: *Workshop on Online Social Networks (WOSN)* (2008), pp. 49–54.

[51] F. Hartmann and I. Baumgart. "SocioPath: Protecting Privacy by Self-Sufficient Data Distribution in User-Centric Networks." In: *Proceedings of the 7th IEEE International Conference on Social Computing and Networking (SocialCom 2014)*. Sydney, Australia, Dec. 2014.

[52] 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, Sept. 2014.

[53] B. Heep. "Effizientes Routing in strukturierten P2P Overlays." PhD thesis. 2011, pp. 49–64.

[54] B. Heep. "R/Kademlia: Recursive and Topology-aware Overlay Routing." In: *Proceedings of 2010 Australasian Telecommunication Networks and Applications Conference (ATNAC 2010)*. Auckland, New Zealand, Nov. 2010, pp. 113–118.

[55] T. Heer et al. "Adapting Distributed Hash Tables for Mobile Ad Hoc Networks." In: *Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)*. IEEE, 2006, pp. 173–178.

[56] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. "Attribute-Based Access Control." In: *Computer* 48.2 (Feb. 2015), pp. 85–88.

[57] B. Huffaker et al. "Topology Discovery by Active Probing." In: *Proceedings of the 2002 Symposium on Applications and the Internet Workshops (SAINT-W '02)*. Nara, Japan, 2002, pp. 90–96.

[58] E. Isaacs et al. “The Character, Functions, and Styles of Instant Messaging in the Workplace.” In: *Proceedings of the 2002 ACM conference on Computer supported cooperative work - CSCW ’02*. New York, New York, USA: ACM Press, 2002, p. 11.

[59] S. Jahid, P. Mittal, and N. Borisov. “EASiER: Encryption-based Access Control in Social Networks with Efficient Revocation.” In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS ’11*. New York, New York, USA: ACM Press, 2011, p. 411.

[60] S. Jahid et al. “DECENT: A Decentralized Architecture for Enforcing Privacy in Online Social Networks.” In: *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE, Mar. 2012, pp. 326–332. arXiv: `arXiv:1111.5377v2`.

[61] S. Jain, K. Fall, and R. Patra. “Routing in a delay tolerant network.” In: *ACM SIGCOMM Computer Communication Review* 34.4 (Oct. 2004), p. 145.

[62] S.-w. S. Jiwon et al. “PrPl: A Decentralized Social Networking Infrastructure.” In: *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond* (2010), 8:1–8:8.

[63] C. Johnson III. *Safeguarding Against and Responding to the Breach of Personally Identifiable Information*. Tech. rep. 2007, p. 7.

[64] J. Klensin. *Simple Mail Transfer Protocol*. Tech. rep. IETF, Oct. 2008.

[65] B. Krishnamurthy and C. E. Wills. “On the Leakage of Personally Identifiable Information via Online Social Networks.” In: *Proceedings of the 2nd ACM workshop on Online social networks*. 2009, pp. 7–12.

[66] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. Tech. rep. RFC Editor, July 2005, pp. 1–32.

[67] N. Lidzborski. *Staying at the forefront of email security and reliability: HTTPS-only and 99.978% availability*. 2014. URL: `http://gmailblog.blogspot.de/2014/03/staying-at-forefront-of-email-security.html` (visited on 11/25/2015).

[68] D. Liu et al. “Confidant: Protecting OSN Data Without Locking It Up.” In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7049 LNCS (2011), pp. 61–80.

[69] E. K. Lua et al. “A Survey and Comparison of Peer-to-Peer Overlay Network Schemes.” In: *IEEE Communications Survey and Tutorial* 7 (2005), pp. 72–93.

[70] M. Lucas and N. Borisov. “FlyByNight: Mitigating the Privacy Risks of Social Networking.” In: *Proceedings of the Seventh ACM Workshop on Privacy in the Electronic Society*. 2008, pp. 1–8.

[71] P. Maymounkov and D. Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric.” In: *Lecture Notes in Computer Science*. Vol. 2429/2002. Cambridge, MA, USA, 2002, pp. 53–65.

[72] G. Mega, A. Montresor, and G. P. Picco. "Efficient Dissemination in Decentralized Social Networks." In: *Proceedings of 2011 IEEE International Conference on Peer-to-Peer Computing (P2P 2011)*. Tokyo, Japan: IEEE, Aug. 2011, pp. 338–347.

[73] G. Mega, A. Montresor, and G. P. Picco. "On Churn and Communication Delays in Social Overlays." In: *2012 IEEE 12th International Conference on Peer-to-Peer Computing (P2P)*. Ieee, Sept. 2012, pp. 214–224.

[74] S. Michiels et al. "Digital Rights Management - A Survey of Existing Technologies." In: *Report CW 428* (2005).

[75] D. Mills et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. Tech. rep. RFC Editor, June 2010, pp. 1–110.

[76] Y. Minsky, A. Trachtenberg, and R. Zippel. "Set Reconciliation with Nearly Optimal Communication Complexity." In: *IEEE Transactions on Information Theory* 49.9 (Sept. 2003), pp. 2213–2218.

[77] S. Nilizadeh et al. "Cachet: A Decentralized Architecture for Privacy Preserving Social Networking with Caching." In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies - CoNEXT '12*. New York, New York, USA: ACM Press, Dec. 2012, p. 337.

[78] P. Olson. *Facebook Closes $19 Billion WhatsApp Deal*. 2014. URL: `https://www.forbes.com/sites/parmyolson/2014/10/06/facebook-closes-19-billion-whatsapp-deal` (visited on 11/03/2017).

[79] T. Paul, A. Famulari, and T. Strufe. "A Survey on Decentralized Online Social Networks." In: *Computer Networks* 75 (Dec. 2014), pp. 437–452.

[80] T. Paul et al. "Exploring Decentralization Dimensions of Social Networking Services : Adversaries and Availability." In: *Proceedings of the First ACM International Workshop on Hot Topics on Interdisciplinary Social Networks Research (HotSocial '12)*. Beijing, China, 2012, pp. 49–56.

[81] 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.

[82] S. Ratnasamy et al. "A Scalable Content-Addressable Network." In: *In Proceedings of ACM SIGCOMM*. San Diego, CA, USA, Aug. 2001, pp. 161–172.

[83] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security*. Tech. rep. Jan. 2012, pp. 1–32.

[84] S. Rhea et al. "Handling Churn in a DHT." In: *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*. Boston, MA, USA, 2004, pp. 127–140.

[85] S. Rhea et al. "OpenDHT: a Public DHT Service and Its Uses." In: *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. Philadelphia, PA, USA: ACM Press, Aug. 2005, pp. 73–84.

[86] S. Roos and T. Strufe. "Dealing with Dead Ends: Efficient Routing in Darknets." In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 1.1 (2016), 4:1–4:30.

[87] A. Rowstron and P. Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems." In: *Lecture Notes in Computer Science*. Vol. 2218/2001. Heidelberg, Germany, Nov. 2001, pp. 329–350.

[88] F. Ruskey and M. Weston. "A Survey of Venn Diagrams." In: *The Electronic Journal of Combinatorics* 5 (2005).

[89] S. C. Rute, J. Crowcroft, and J. Kempf. "User-Centric Networking." In: *Proceedings of Dagstuhl Seminar 10372*. Ed. by P. Mendes. Wadern, Germany, 2010.

[90] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. Tech. rep. IETF, Mar. 2011.

[91] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. Tech. rep. Mar. 2011.

[92] M. Sayrafiezadeh. "The Birthday Problem Revisited." In: *Mathematics Magazine* 67.3 (June 1994), p. 220.

[93] B. Schneier. *Applied Cryptography*. 1st ed. John Wiley & Sons, 2015, pp. 1–784.

[94] L. Schwittmann et al. "Privacy Preservation in Decentralized Online Social Networks." In: *IEEE Internet Computing* 18.2 (2014), pp. 16–23.

[95] L. Schwittmann et al. "SoNet - Privacy and Replication in Federated Online Social Networks." In: *Proceedings - International Conference on Distributed Computing Systems* (2013), pp. 51–57.

[96] T. M. Shafaat, A. Ghodsi, and S. Haridi. "Handling Network Partitions and Mergers in Structured Overlay Networks." In: *Peer-to-Peer Computing, 2007. P2P 2007. Seventh IEEE International Conference on*, pp. 132–139.

[97] A. Shakimov et al. "Vis-à-Vis: Privacy-Preserving Online Social Networking via Virtual Individual Servers." In: *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*. IEEE, Jan. 2011, pp. 1–10.

[98] C. E. Shannon. "Prediction and Entropy of Printed English." In: *Bell System Technical Journal* 30.1 (Jan. 1951), pp. 50–64.

[99] R. Sharma and A. Datta. "SuperNova: Super-peers based architecture for decentralized online social networks." In: *2012 Fourth International Conference on Communication Systems and Networks (COMSNETS 2012)*. IEEE, Jan. 2012, pp. 1–10. arXiv: `arXiv:1105.0074v2`.

[100] W. Sherchan, S. Nepal, and C. Paris. "A Survey of Trust in Social Networks." In: *ACM Computing Surveys* 45.4 (Aug. 2013), pp. 1–33.

[101] M. Shiels. *The anti-Facebook*.

[102] A. Singh et al. "Eclipse Attacks on Overlay Networks: Threats and Defenses." In: *25th IEEE International Conference on Computer Communications ({INFOCOM} 2006)*. Barcelona, Spain, Apr. 2006.

[103] *Skype*. URL: `https://skype.com` (visited on 10/31/2014).

[104] *Snapchat*. URL: `https://www.snapchat.com` (visited on 11/22/2015).

[105] R. Sofia et al. "User centric networking and services: Part I [Guest Editorial]." In: *IEEE Communications Magazine* 52.9 (Sept. 2014), p. 18.

[106] I. Stoica et al. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." In: *Proceedings of the {ACM} {SIGCOMM} '01 Conference*. San Diego, California, Aug. 2001.

[107] J. Su et al. "Haggle: Seamless networking for mobile applications." In: *Lecture Notes in Computer Science* 4717.2007 (2007), pp. 391–408.

[108] *Telegram*. URL: `https://telegram.org/` (visited on 07/29/2015).

[109] *TextSecure*. URL: `https://whispersystems.org` (visited on 07/29/2015).

[110] The Guardian. *WhatsApp privacy backlash: Facebook angers users by harvesting their data*. 2016. URL: `https://www.theguardian.com/technology/2016/aug/25/whatsapp-backlash-facebook-data-privacy-users` (visited on 03/11/2017).

[111] *Threema*. URL: `https://threema.ch` (visited on 07/29/2015).

[112] D. Tsolis et al. *Digital Rights Management for E-Commerce Systems*. IGI Global, 2008, p. 38.

[113] J. Ugander et al. "The Anatomy of the Facebook Social Graph." In: (Nov. 2011), p. 17. arXiv: `1111.4503`.

[114] B. Viswanath et al. "On the Evolution of User Interaction in Facebook." In: *Proceedings of the 2nd ACM workshop on Online social networks - WOSN '09* (2009), p. 37.

[115] W. Vogels. "Eventually Consistent." In: *Queue* 6.6 (Oct. 2008), p. 14.

[116] *Whatsapp*. URL: `http://www.whatsapp.com` (visited on 10/31/2014).

[117] X. Xing et al. "Routing in User-Centric Networks." In: *IEEE Communications Magazine* 52.9 (Sept. 2014), pp. 44–51. arXiv: `14 [0163-6804]`.

[118] Z. Yao et al. "Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks." In: *Proceedings of the 2006 IEEE International Conference on Network Protocols* (Nov. 2006), pp. 32–41.

[119] B. Y. Zhao et al. "Tapestry: A Resilient Global-Scale Overlay for Service Deployment." In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53.